KERROSARKKITEHTUURIN SUUNNITTELUMALLIT

Kuisma Lehtonen

15.8.2006
Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

TIIVISTELMÄ

Suunnittelumallit ovat yleisiä ratkaisuja tiettyihin oliopohjaisiin ohjelmistosuunnittelun ongelmiin. Suunnittelumallit auttavat ohjelmistosuunnittelijaa tai ohjelmiston arkkitehtuurin suunnittelijaa löytämään oikean ratkaisun nopeammin. Malli kuvaa ongelman ja sen ratkaisun abstraktilla tasolla. Ratkaisu ei ole yksittäinen ja konkreettinen, vaan se on runko, jota voi soveltaa monella eri tavalla. Ratkaisu kertoo, millaisella luokkien ja olioiden yleisrakenteella ongelma ratkaistaan. Suunnittelumallia sanotaan arkkitehtoniseksi suunnittelumalliksi, kun se tukee arkkitehtonista suunnittelua. Arkkitehtoniset suunnittelumallit tarjoavat perustavanlaatuisia rakenteellisia ratkaisuja ohjelmistolle. Arkkitehtonisten suunnittelumallien avulla suunnitellaan esimerkiksi järjestelmän kerrostaminen, hajauttaminen, komponenttien välinen kommunikointi ja yhteistyö, sekä varaudutaan järjestelmän tulevaisuuden haasteisiin kuten laajennettavuuteen ja mukautuvuuteen. Tässä pro gradu –tutkielmassa paneudutaan pääasiassa J2EE-teknologiaan arkkitehtonisten suunnittelumallien esittelyyn ja niiden toteutusstrategioihin. Tutkielmaan liittyy J2EE-arkkitehtuuriin perustuva ja esiteltyjä J2EE-suunnittelumalleja hyväksikäyttävä palvelinsovellus. Palvelinsovelluksessa käytetään hyväksi kirjallisuudessa kuvattuja ja tutkielmassa esiteltyjä suunnittelumallien toteutusstrategioita.

Avainsanat: suunnittelumalli, arkkitehtuuri, arkkitehtoninen suunnittelumalli, J2EE

SISÄLLYSLUETTELO

1 JOHDANTO

Toivosen (2001) mukaan suunnittelumallit ovat yleisiä ratkaisuja tiettyihin oliopohjaisiin ohjelmistosuunnittelun ongelmiin. Ratkaisut ovat kehittyneet ja jalostuneet pitkän ajan kuluessa, joten ne eivät ole sellaisia, joita ihmisillä on tapana ensimmäiseksi kokeilla. Mallin taustalla on lukuisia suunnittelu- ja uudelleenohjelmointikierroksia, joilla on pyritty parantamaan ohjelmiston uudelleenkäytettävyyttä ja joustavuutta. Ns. yleisiä oliopohjaisia suunnittelumalleja on dokumentoitu 1990-luvun alusta saakka. Suunnittelumallissa ratkaisu on ytimekäs ja helposti sovellettavassa muodossa. Koskimiehen (2003) mukaan suunnittelumallit auttavat suunnittelijaa löytämään oikean ratkaisun nopeammin. Malli kuvaa ongelman ja sen ratkaisun abstraktilla tasolla. Ratkaisu ei ole yksittäinen ja konkreettinen, vaan se on runko, jota voi soveltaa monella eri tavalla. Ratkaisu kertoo, millaisella luokkien ja olioiden yleisrakenteella ongelma ratkaistaan. Ohjelmakoodin avulla voidaan antaa vain esimerkkejä suunnittelumallista. Suunnittelumallien käyttäminen järjestelmässä ei sinänsä tuo lisäarvoa, vaan suunnittelumallin käyttämisen tulee aina lähteä ongelman tunnistamisesta. Suunnittelumalleja ei pidä käyttää kritiikittömästi. Esimerkiksi mallin joustavuus ja muunneltavuus lisäävät epäsuoraa ohjausta, mikä voi monimutkaistaa suunnitteluratkaisua ja vähentää suorituskykyä. Suunnittelumallia tulee käyttää vain silloin kun sen tarjoamaa joustavuutta todella tarvitaan.

Suunnittelumallit tarjoavat suunnittelijoille yhteisen sanaston, jota voi käyttää keskusteluissa, dokumentoinnissa ja vaihtoehtoisen ratkaisumallin etsinnässä. Suunnittelumallien avulla järjestelmästä voidaan keskustella korkeammalla abstraktiotasolla. Suunnittelumallien tunteminen helpottaa olemassaolevien järjestelmien ymmärtämistä, sillä Toivosen (2001) mukaan useimmat laajat oliopohjaiset järjestelmät käyttävät suunnittelumalleja. Erityisen hyödyllisiä suunnittelumallit ovat siinä vaiheessa, kun määrittelyssä muodostettu malli muutetaan toteutusmalliksi. Määrittelyssä syntyneitä malleja on suunniteltava usein uudestaan uudelleenkäytettävyyden saavuttamiseksi. Suunnittelumallit käsittelevät näitä näkökohtia ja siksi niitä sanotaankin suunnittelumalleiksi.

Ohjelmiston arkkitehtuuri on sen keskeisten osien ja niiden välisten staattisten ja dynaamisten suhteiden kuvausta korkealla abstraktiotasolla (Koskimies, 2000). Arkkitehtuuriin kuuluu tekniikoita, kuvaustapoja ja standardeja ratkaisuja, joita tarvitaan ohjelmistojen korkean tason suunnittelussa. Arkkitehtuuri määrittelee miten haluttu toiminnallisuus saadaan aikaan. Arkki-

tehtuurisuunnittelussa suunnitellaan järjestelmän kerrokset, merkittävät komponentit, arkkitehtuurityylit, mahdollinen kehysarkkitehtuurin ydin, ohjelmiston sijoittelu laitteistoihin, ohjelmistoalustat (tietokantaohjelmistot, tietoliikenneohjelmistot jne.), prosessit ja niiden kommunikointi, käyttöliittymäratkaisut ja korkean tason suunnittelumallit, sekä muut keskeiset ohjelmiston arkkitehtuuriin vaikuttavat ratkaisut. Suunnitelma kuvataan erilaisilla kaavioilla, esim. luokkakaavioilla, komponenttikaavioilla, sijoittelukaavioilla ja sekvenssikaavioilla.

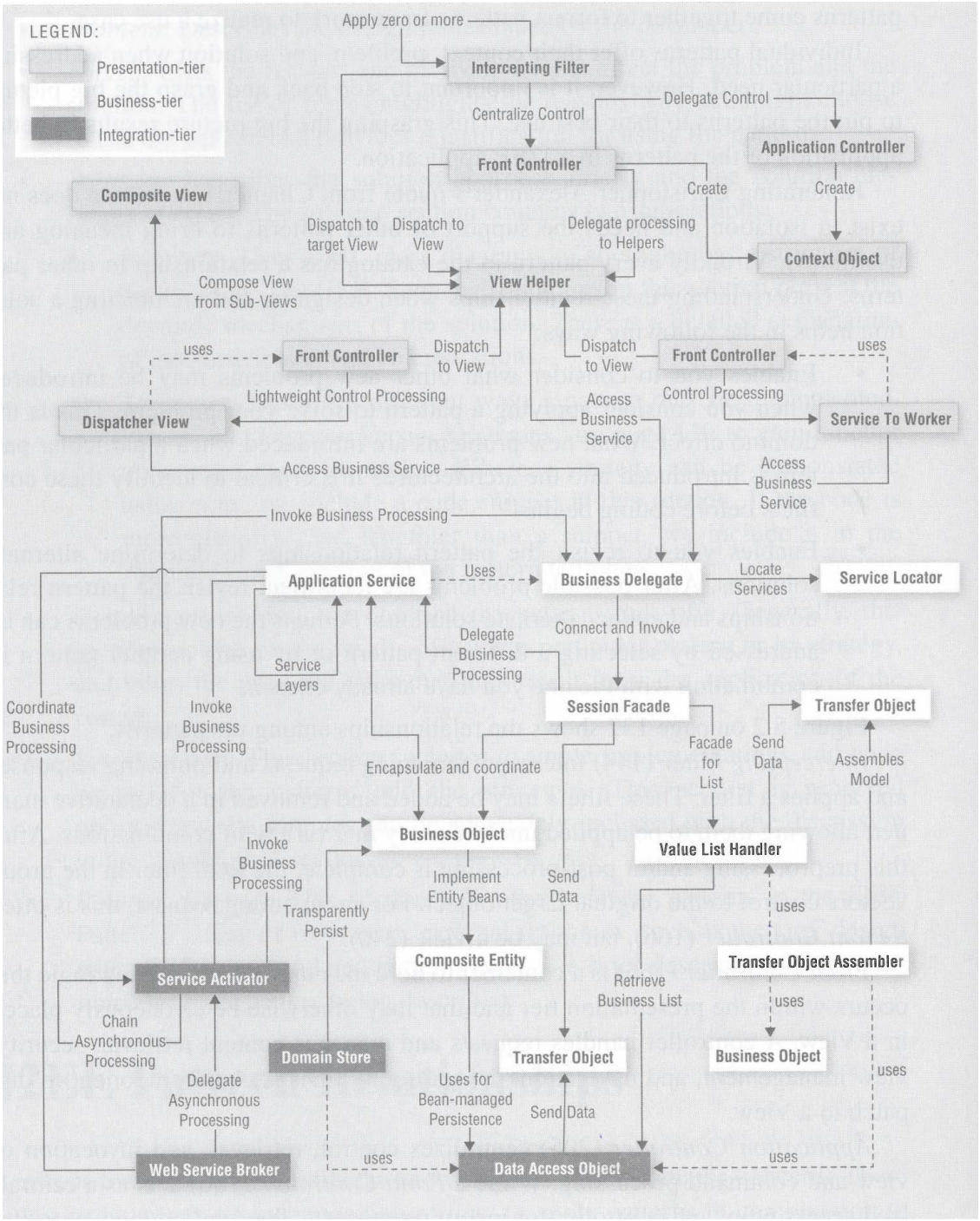
Kerroksittainen ohjelmistoarkkitehtuuri tarkoittaa ohjelmiston rakennetta, joka koostuu loogisesti yhtenäisistä eri käsitetasoilla olevista kerroksista siten, että ylempi kerros käyttää alemman kerroksen palveluja (Koskimies, 2000). Alempi kerros tarjoaa aina yleisempiä palveluja kuin ylempi kerros ja siten ylempi kerros lähenee sovellusaluetta. Järjestelmä voi esimerkiksi koostua kolmesta kerroksesta; tietokannan abstrahointikerros (integrointikerros), sovellusalueen käsitteiden toteutuskerros (liiketoimintakerros) ja käyttöliittymäkerros (esityskerros). Tällöin puhutaan kolmikerrosarkkitehtuurista. Tavoitteena kerrosarkkitehtuurissa on, että kutsurakenne on hierarkkinen ja ohitukseton. Hierarkkisessa rakenteessa kutsut kulkevat aina joko samalla tasolla tai korkeammalta tasolta alemmalle tasolle, mutta eivät alemmalta tasolta ylemmälle tasolle. Ohituksettomassa rakenteessa kutsut eivät ohita kerroksia, vaan kohdistuvat aina välittömästi seuraavaan kerrokseen. Kerroksittainen rakenne mahdollistaa ohjelmiston kehittämisen ja ymmärtämisen käsitetasoltaan kohoavissa eli sovellusalueen käsitteistöä lähentyvissä kerroksissa. Kerrosarkkitehtuuri vähentää ohjelmanosien riippuvuutta toisistaan ja siten parantaa ylläpidettävyyttä ja muunneltavuutta. Toisaalta huonosti suunniteltu kerrosarkkitehtuuri voi enemmän haitata ohjelmiston kehittämistä kuin auttaa siinä.

Suunnittelumalleja jäsennetään erilaisten luokitustapojen perusteella. Toivonen (2001) luokittelee läheisesti yhteen kuuluvat suunnittelumallit perheiksi käyttötarkoituksen ja kohteen perusteella. Käyttötarkoitus kuvaa mallin toimintaa. Luontimallit käsittelevät olioiden luontiprosessia, rakennemallit koskevat luokkien ja olioiden koosteita ja käyttäytymismallit käsittelevät luokkien ja olioiden vuorovaikutustapoja ja vastuunjakamista. Kohde kertoo, liittyykö malli ensisijaisesti luokkiin vai olioihin. Buschmann & al. (1996) luokittelevat mallit arkkitehtonisiin malleihin, suunnittelumalleihin ja idiomeihin eli tiettyyn ohjelmointikieleen liittyviin malleihin. Arkkitehtoniset mallit tarjoavat Buschmannin & al. (1996) mukaan perustavanlaatuisia rakenteellisia ratkaisuja ohjelmistolle. Ne muodostavat määriteltyjä alijärjestelmiä, niiden vastuita, sääntöjä ja ohjeita niiden välisten suhteiden muodostamiseksi. Suunnittelumallit esittävät ratkaisun usein esille tulevaan ongelmaan tietyssä kontekstissa. Idiomit ovat alem-

man tason malleja, jotka esittävät ratkaisun tietynlaisen komponentin toteuttamiseksi tai komponenttien välisten suhteiden muodostamiseksi tietyssä ohjelmointikielessä. Alur & al. (2003) luokittelevat mallit sen mukaan minne ne sijoittuvat ohjelmiston arkkitehtuurissa. Suunnittelumallit esitetään kolmikerrosarkkitehtuurissa, jossa ne voivat olla joko esitys-, liiketoiminta- tai integrointikerroksen suunnittelumalleja. Aina suunnittelumallien dokumentoijat eivät ole samaa mieltä miten jokin suunnittelumalli pitäisi luokitella. Esim. Maciaszek & al. (2005) kertovat Julkisivun (Facade) olevan arkkitehtoninen suunnittelumalli, kun taas Buschmannin & al. (1996) mukaan se on vain tavallinen suunnittelumalli.

Arkkitehtoninen suunnittelumalli liittyy arkkitehtuuritason ratkaisuihin. Buschmannin & al. (1996) mukaan arkkitehtonisen suunnittelumallin avulla suunnitellaan esimerkiksi järjestelmän kerrostaminen, hajauttaminen, komponenttien välinen kommunikointi ja yhteistyö, sekä varaudutaan järjestelmän tulevaisuuden haasteisiin, kuten laajennettavuuteen ja mukautuvuuteen. Suunnittelumallia sanotaan arkkitehtoniseksi suunnittelumalliksi, kun se tähtää arkkitehtoniseen suunnitteluun (Maciaszek & al., 2005). Esimerkiksi Alurin & al. (2003) kuvaamassa arkkitehtuurissa järjestelmä on jaoteltu kolmeen eri käsitetasolla olevaan kerrokseen (ks. kuva 1.1) ja esityskerroksen komponentti tarvitsee esitysmallin, joka muodostetaan alempien kerrosten komponenttien avulla. Jotta hierarkkinen ja ohitukseton kutsurakenne toteutuisi, esitystason komponentti ei suoraan hae tietoja tietokannasta, vaan monien eri suunnittelumallien muodostaman arkkitehtuurin avulla tekee palvelupyyntöjä aina välittömästi seuraavalle, alemmalla tasolla olevalle kerrokselle. Pyynnön toteutumiseen osallistuu tässä tapauksessa ainakin liiketoimintakerroksesta Liiketoimintadelegaatti, Palvelunpaikallistaja, Sessiojulkisivu, Sovelluspalvelu, Kontekstiolio, Komposiittientiteetti, Kuljetusolio, Kuljetusolion kokooja, integrointikerroksesta Tiedonhakuolio ja käyttötapaukseen liittyvät esityskerroksen suunnittelumallit.

Tässä pro gradu-tutkielmassa paneudutaan pääasiassa J2EE-teknologiaan perustuvien suunnittelumallien esittelyyn ja niiden toteutusstrategioihin. J2EE-suunnittelumallit muodostavat J2EE-arkkitehtuurin. Alurin & al. (2003) mukaan J2EE-teknologiaan perustuvia suunnittelumalleja käytetään usein yhdessä ratkaisemaan arkkitehtonisia ongelmia, kuten esimerkiksi kerrostaminen, sovelluksen tärkeimpien komponenttien suunnitteleminen ja kommunikointi komponenttien ja kerrosten välillä.



Kuva 1.1: J2EE-teknologiaan perustuvat suunnittelumallit J2EE-arkkitehtuurissa (Alur & al., 2003)

Sun Java Centerin arkkitehdit alkoivat nimeämään ja dokumentoimaan J2EE-teknologiaan perustuvia suunnittelumalleja vuodesta 1999 alkaen. J2EE-suunnittelumallit pohjautuvat jo aiemmin esiteltyihin, ns. yleisiin oliopohjaisiin suunnittelumalleihin. Yhdessä J2EE-suunnittelumallissa on saatettu käyttää hyväksi useampaa jo aiemmin esiteltyä suunnittelumallia so-

veltaen J2EE-teknologiaa tai esitellään suunnittelumalli, joka ei perustu ainakaan suoranaisesti millekään aiemmalle mallille. Tässä tutkielmassa esitellään joitakin ”vanhempia” tai yleisesti tunnettuja suunnittelumalleja ja useampia J2EE-teknologiaan perustuvia arkkitehtonisia suunnittelumalleja. Lisäksi J2EE-suunnittelumalleille esitellään yksi tai muutama toteutusstrategia, jotka on kuvattu kirjallisuudessa. Tutkielmassa ei paneuduta arkkitehtuurin suunnitteluun sinänsä, vaan ainoastaan siltä osin mikä koskee suunnittelumalleja. J2EE-suunnittelumallit esitellään kolmikerrosarkkitehtuuriin perustuvassa järjestelmässä, johon kuuluvat esityskerros, liiketoimintakerros ja integrointikerros.

Tähän pro gradu-tutkielmaan liittyy J2EE-arkkitehtuuriin perustuva ja esiteltyjä J2EE-suunnittelumalleja hyväksikäyttävä yksinkertainen palvelinsovellus (ks. liitteet 1-7). Esimerkkisovelluksessa käytetään hyväksi kirjallisuudessa kuvattuja ja tutkielmassa esiteltyjä suunnittelumallien toteutusstrategioita. Sovellusta voi käyttää WWW-selaimen avulla. Sovellus on Kurssinhallintasovellus, joka hakee tietokannasta kursseja, sekä niihin liittyviä opettajia ja opiskelijoita. Kurssin tilan voi päivittää aloitetuksi ja päättyneeksi. Kurssiin voi liittää luentomonisteen tiedostona, joka ladataan WWW-sivulta ja tallennetaan palvelimen hakemistoon. Kaikkia J2EE-suunnittelumalleja ei tämän tutkielman puitteissa voida toteuttaa sovelluksessa, vaan sovellus rajoittuu arkkitehtuuriltaan esityskerroksen komponentteihin, jotka muodostavat loogisen kokonaisuuden. Liiketoiminta- ja integrointikerros on esimerkkisovelluksessa korvattu apuluokalla, joka muodostaa tietokantayhteyden.

Tutkielman tavoitteena on esitellä J2EE-teknologiaan perustuvia suunnittelumalleja, ja joitakin yleisiä suunnittelumalleja, joita voidaan käyttää yhdessä J2EE-suunnittelumallien kanssa. Tarkoituksena on antaa yleiskuva olemassaolevista J2EE-suunnittelumalleista, niiden käyttötarkoituksesta ja soveltamistavoista. Suunnittelumallien vaikutukset ja toteutusstrategiat kootaan kirjallisuudesta. Kaikkia kirjallisuudessa esiteltyjä toteutusstrategioita ei ole esitelty, vaan mahdollisuuksien mukaan on valittu toisiaan tukevia strategioita. Liiketoimintakerroksen suunnittelumallien strategioista on pyritty pääsääntöisesti valitsemaan niitä, jotka tukevat EJB-teknologian käyttämistä. EJB-teknologia on eräs vakiintuneista yrityssovellusteknologioista. Esityskerroksen suunnittelumallien strategioista on pyritty esittelemään ainakin Kurssinhallintasovelluksessa toteutetut strategiat. J2EE-suunnittelumallien keskeisenä lähteenä on käytetty Alur & al. (2003). Suunnittelumallien toteutettavuutta arvioidaan toteuttamalla esityskerroksen suunnittelumallit tutkielmaan liittyvässä sovelluksessa (liitteet 1-7).

Tutkielman luvussa 2 esitellään yleisiä suunnittelumalleja, joihin viitataan tutkielman muissa osissa. Osa näistä suunnittelumalleista on arkkitehtonisia ja se on mainittu kyseisen mallin esittelyssä. Jotkin näistä malleista kuuluvat kiinteästi toisiinsa ja niitä käytetään tutkielmaan liittyvässä Kurssinhallintasovelluksessa osana J2EE-suunnittelumallia. Luvussa 3 esitellään tutkielman varsinaisena kohteena olevia J2EE-suunnittelumalleja. J2EE-suunnittelumallit on jaoteltu kolmikerrosarkkitehtuurin mukaisesti esitys,- liiketoiminta- ja integrointikerroksien suunnittelumalleihin. Jokaisesta suunnittelumallista kerrotaan kuvaus, käyttötarkoitus, vaikutukset, toteutusstrategioita ja kuvataan suunnittelumallin soveltamista Kurssinhallintasovelluksessa toteutettujen esityskerroksen suunnittelumallien osalta. Liiketoiminta- ja integrointikerroksen toteutusstrategioissa on kerrottu esimerkkejä miten kyseistä suunnittelumallia voisi soveltaa Kurssinhallintasovelluksessa. Luvun 4 yhteenvedossa on koostettu suunnittelumallien käytön hyviä ja huonoja puolia ja kerrottu kokemuksia sovelluksen toteuttamisesta. Sovellus on dokumentoitu liitteissä 1-7. Liitteisiin viitataan kolmannessa luvussa olevien esityskerroksen J2EE-suunnittelumalleissa, joita on käytetty Kurssinhallintasovelluksessa.

2 YLEISIÄ SUUNNITTELUMALLEJA

Tässä luvussa esitellään joitakin yleisimpiä oliopohjaisia ja arkkitehtonisia suunnittelumalleja, joihin viitataan myöhemmissä luvuissa. Suunnittelumalleja voidaan käyttää yhdessä J2EE-suunnittelumallien kanssa. Suunnittelumalleja Abstrakti tehdas, Tehdasmetodi ja Komento, on käytetty tutkielmaan liittyvässä Kurssinhallintasovelluksessa osana J2EE-suunnittelumallia. Toteutus on kuvattu kyseisen J2EE-suunnittelumallin esittelyssä.

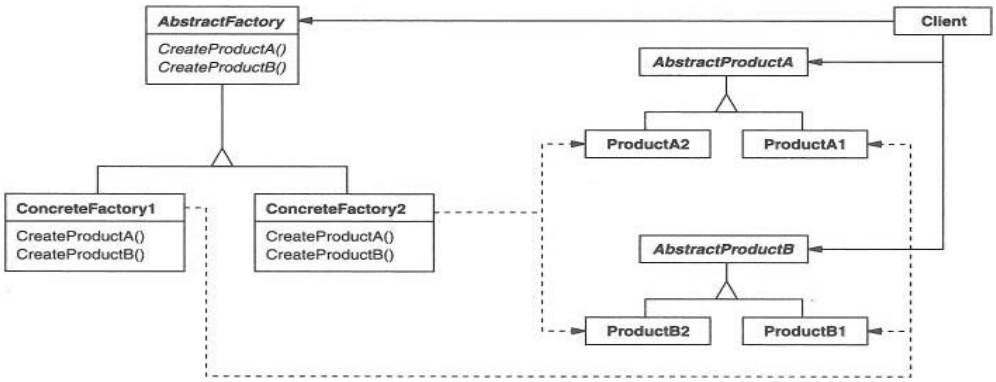
2.1 Abstrakti tehdas (Abstract Factory)

Abstrakti tehdas tuottaa rajapinnan, jolla luodaan toisiinsa liittyvien olioiden muodostamia olioperheitä määrittelemättä olioiden konkreettisia luokkia (Toivonen, 2001). Abstrakti tehdas määrittelee konkreettisten luokkien luontioperaatiot abstraktien luokkien avulla. Asiakassovellukset saavat tarvitsemansa olion abstraktilta tehtaalta tuntematta konkreettisia luokkia. Asiakassovellukset käyttävät saamaansa oliota sen rajapinnan esittelemien metodien kautta. Abstrakti tehdas on arkkitehtoninen suunnittelumalli, jonka avulla sovellus voi toimia eri tavoin, riippuen rajapinnan taakse piilotetusta kutsuttavasta olioperheestä (Maciaszek & al., 2005).

Toivosen (2001) mukaan Abstraktia tehdasta käytetään, kun

- sovelluksen tulee olla riippumaton siitä miten sen tuotteet luodaan, koostetaan ja esitetään
- sovelluksessa on käytössä on useita tuoteperheitä ja sovellus halutaan konfiguroida käyttämään yhtä niistä
- perheeseen kuuluvia tuoteolioita on tarkoitus käyttää ja yhdistellä vain keskenään ja ratkaisun on valvottava tätä rajoitusta
- halutaan luoda tuotteista luokkakirjasto, jonka toteutustapaa ei paljasteta.

AbstractFactory (kuva 2.1) määrittelee abstraktien tuoteolioiden luontioperaatioiden kutsumuodon ja siirtää olioden luonnin ConcreteFactory-aliluokalle. ConcreteFactory toteuttaa konkreettisten tuoteolioiden luontioperaatiot. Tavallisesti ajon aikana luodaan yksi ilmentymä ConcreteFactory-luokasta. Tämä konkreettinen tehdas luo tietyn tuoteperheen tuoteolioita. Jos tarvitaan muita tuoteolioita, on käytettävä toista konkreettista tehdasta.



Kuva 2.1: Abstrakti tehdas (Toivonen, 2001)

AbstractProduct määrittelee tietyn tyyppisen tuotteen rajapinnan. Product kuvaa tuoteolion, jonka vastaava konkreettinen tehdas luo. Se toteuttaa AbstractProduct-rajapinnan. Product-aliluokkien tehtävänä on todella luoda tuotteet. Tavallisin toteutustapa on määritellä jokaista tuotetta kohden ns. Tehdasmetodi (ks. Tehdasmetodi-malli). Client käyttää vain AbstractFactory- ja AbstractProduct-luokkien määrittämiä rajapintoja.

Taulukossa 2.1 on kuvattu mallin vaikutuksia Toivosen (2001) perusteella.

Taulukko 2.1: Abstraktin tehtaan vaikutuksia.

Vaikutus	Selitys
Eristää konkreettiset luokat	Abstraktin tehtaan avulla voi kontrolloida, mitä luokkia sovellus luo. Tehdas kapseloi tuoteolion luontiprosessin ja siten eristää sovellukset toteutusluokista. Sovellus käyttää tuoteilmentymiä niiden abstraktien rajapintojen kautta.
Helppo vaihtaa tuoteperheitä	Konkreettisen tehtaan luokka esiintyy sovelluksessa alustettaessa, eli vain kerran. Tuoteperheen vaihtaminen tapahtuu vaihtamalla konkreettista tehdasta.
Valvoo tuotteiden käytön yhtenäisyyttä	Abstrakti tehdas valvoo että sovellus käyttää olioita vain yhdestä tuoteperheestä kerrallaan, koska tuoteperheen tuotteiden on toimittava yhdessä.
Uudenlaisten tuotteiden lisääminen on vaikeaa	AbstractFactory-rajapinta (kuva 2.1) kiinnittää tuotejoukon, joka voidaan luoda. Uudenlaisten tuotteiden lisääminen vaatii rajapinnan laajentamista, mikä puolestaan vaatii AbstractFactory-luokan ja sen aliluokkien muuttamista vastaamaan rajapintaa.

2.2 Välittäjä (Broker)

Välittäjää voidaan Buschmannin & al. (1996) mukaan käyttää hajautettujen järjestelmien välisessä kommunikoinnissa. Tämä arkkitehtoninen suunnittelumalli vastaa kommunikoinnin ohjaamisesta toisistaan riippumattomien komponenttien välillä, kuten saapuneiden pyyntöjen välittämisestä eteenpäin ja vastauksien palauttamisesta kutsuvalle komponentille.

Buschmannin & al. (1996) mukaan Välittäjää käytetään, kun

- komponentit käyttävät toisten komponenttien tarjoamia palveluita käyttäen etäpyyntöjä tietämättä missä komponentit sijaitsevat
- komponentteja tulee pystyä vaihtamaan, lisäämään tai poistamaan järjestelmien ollessa käynnissä
- arkkitehtuurin tulee piilottaa järjestelmä- ja toteutusriippuvaiset yksityiskohdat komponentteja ja palveluita käyttäviltä asiakassovelluksilta.

Välittäjien avulla saadaan Buschmannin & al. (1996) mukaan aikaiseksi löyhempiä liitoksia asiakassovelluksien ja palvelinsovelluksien välille. Asiakassovellukset pääsevät käyttämään palvelimella olevia palveluita välittäjän kautta lähettämällä palvelupyynnön välittäjälle. Välittäjän tehtävänä on paikallistaa sovelias palvelin, jolle se välittää palvelupyynnön ja palauttaa palvelun tuottamat tulokset tai virheilmoitukset takaisin kutsuvalle asiakassovellukselle. Välittäjän avulla vältetään sisäiset prosessikutsut alempiin kerroksiin ja mahdollistetaan alusta- ja ohjelmointikielestä riippumaton kommunikointitapa järjestelmien välille.

Taulukossa 2.2 on kuvattu mallin vaikutuksia Buschmannin & al. (1996) perusteella.

Taulukko 2.2: Välittäjän vaikutuksia.

Vaikutus	Selitys
Sijainnin läpinäkyvyys	Välittäjä vastaa palvelimen paikallistamisesta yksilöllisen tunnisteen avulla. Asiakassovellusten ei tarvitse tietää missä palvelimet sijaitsevat. Samoin myöskään palvelimet eivät tiedä, missä asiakassovellukset ovat, vaan saavat kaikki palvelupyyntönsä aina paikalliselta Välittäjä-komponentilta.
Muutettavuus ja laajennettavuus paranee	Palvelinta ja asiakassovellusta voidaan muuttaa toisistaan riippumatta, jos niiden rajapintaa ei tarvitse muuttaa. Välittäjää voi muuttaa sisäisesti sen vaikuttamatta palvelimeen ja asiakassovellukseen, jos Välittäjän API ei muutu.
Siirrettävyys helpottuu	Välittäjä piilottaa alla olevan käyttöjärjestelmän ja tietoliikenneverkon tekniset yksityiskohdat asiakassovellukselta. Järjestelmän siirrettävyys helpottuu, koska koko järjestelmää ei tarvitse muuttaa allaolevan ympäristön vaihtuessa.
Uudelleenkäytettävyys paranee	Osaa järjestelmän palveluista voidaan uudelleenkäyttää Välittäjien avulla. Esim. pääsy tietokantaan ja tulostuspalvelut voidaan ottaa käyttöön uudessa järjestelmässä käyttäen samaa Välittäjä-komponenttia.
Suorituskyky heikkenee	Välittäjää käyttävät sovellukset ovat hitaampia kuin sovellukset, jotka käyttävät suoraan palvelimen palveluita. Suoraan palvelimen palveluita käyttävät asiakassovellukset ovat kuitenkin riippuvaisia allaolevasta tekniikasta, eivätkä siten ole niin joustavia, helposti siirrettävissä olevia ja muutettavia kuin Välittäjää käyttävät sovellukset.

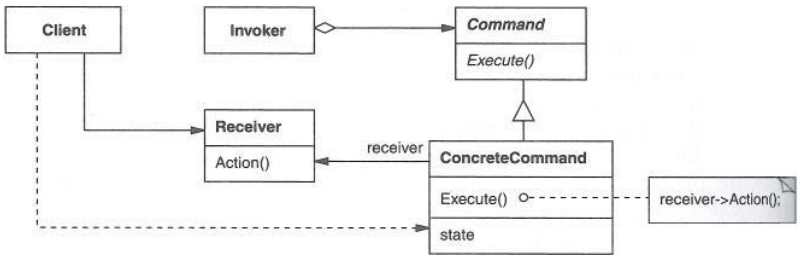
2.3 Komento (Command)

Komento-suunnittelumallissa oliolle lähetetään pyyntö tietämättä suoritettavasta operaatiosta tai pyynnön vastaanottajasta (Toivonen, 2001). Abstrakti Command-luokka (kuva 2.2) määrittelee rajapinnan operaatioiden luomiseksi. Jonkin olion tekemä pyyntö muutetaan olioksi, joka välitetään edelleen abstraktin Command-luokan toteuttavalle konkreettiselle oliolle. Konkreettinen Command-aliluokka määrittelee vastaanottaja-toiminto-parin, tallettamalla vastaanottajan ilmentymämuuttujaan ja toteuttamalla operaation, joka herättää vastaanottajan. Vastaanottaja suorittaa pyynnön operaation. Suunnittelumallin idea on, että operaation käynnistävä olio erotetaan siitä oliosta, joka tietää, miten operaatio pitää suorittaa.

Suunnittelumallia käytetään Toivosen (2001) mukaan, kun

- suoritettava pyyntö halutaan antaa parametrina
- pyyntöjä määritellään, jonotetaan ja suoritetaan eri aikoina
- pyyntöjä halutaan perua
- halutaan pitää muutoksista lokitietoa; lokin avulla pyynnöt voidaan toistaa esimerkiksi järjestelmän kaaduttua
- halutaan suorittaa useita operaatioita kerralla; operaatioita voi kutsua samalla tavalla yhteisen rajapinnan avulla ja siten muodostaa alkeisoperaatioista koottuja korkean tason operaatioita.

Command (kuva 2.2) määrittelee suoritettavan eli konkreettisen operaation rajapinnan. ConcreteCommand sitoo yhteen vastaanottajan (Receiver) ja toiminnon (Action) ja toteuttaa Execute()-operaation kutsumalla vastaavaa Receiver-operaatiota. Client luo ConcreteCommand-olion ja asettaa sille vastaanottajan. Invoker käskee komentoa toteuttamaan pyynnön. Receiver tietää miten pyyntöön liittyvä operaatio suoritetaan. Receiver voi olla mikä tahansa luokka.



Kuva 2.2: Komento (Toivonen, 2001)

Taulukossa 2.3 on kuvattu mallin vaikutuksia Toivosen (2001) perusteella.

Taulukko 2.3: Komennon vaikutuksia.

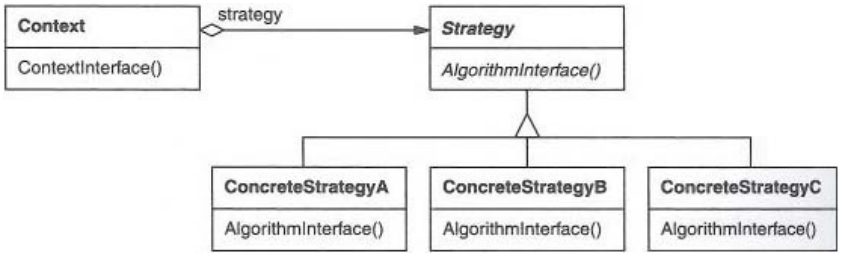
Vaikutus	Selitys
Olio ei tiedä operaation toteuttajasta	Operaation käynnistävä olio erotetaan siitä oliosta, joka tietää, miten operaatio pitää suorittaa.
Laajennettavuus ja muokattavuus paranee	Komennot ovat olioita, joita voi muokata ja laajentaa. Uusia komentoja voi lisätä helposti.

2.4 Strategia (Strategy)

Strategia-suunnittelumalli määrittelee algoritmiperheen, kapseloi kunkin algoritmin ja tekee niistä keskenään vaihdettavia (Toivonen, 2001). Algoritmia voidaan muuttaa muuttamatta sitä käyttävää sovellusta. Strategian avulla luokkaan saadaan liitettyä erilaisia käyttäytymistapoja.

Toivosen (2001) mukaan suunnittelumallia käytetään, kun

- joukko toisiinsa liittyviä luokkia eroaa toisistaan vain käyttäytymiseltään
- algoritmista tarvitaan erilaisia variaatioita
- sovelluksen ei pidä tietää algoritmin käyttämistä tietorakenteista
- luokan käyttäytymisen variaatiot ilmenevät sen operaatioissa ehtolausekkeina, jotka suunnittelumallissa sijoitetaan omiin strategialuokkiinsa.



Kuva 2.3: Strategia (Toivonen, 2001)

Strategy (kuva 2.3) määrittelee kaikille algoritmeille yhteisen rajapinnan. ConcreteStrategy toteuttaa yksittäisen algoritmin Strategy-rajapinnan mukaisesti. Context käyttää Strategyn määrittelemää rajapintaa kutsuakseen algoritmia, joka on määritelty ConcreteStrategy-luokassa. Context pitää viitettä Strategy-olioon, ja voi tarjota rajapinnan, jonka avulla Strategy pääsee käsiksi Context-olion tietoihin (ContextInterface). Context välittää myös asiakkailtaan tulevat pyynnöt strategioilleen.

Taulukossa 2.4 on kuvattu mallin vaikutuksia Toivosen (2001) perusteella.

Taulukko 2.4: Strategian vaikutuksia.

Vaikutus	Selitys
Yhteen liittyvien algoritmien perheet	Strategy-luokkien hierarkiat määrittelevät algoritmiperheen käyttäytymisjoukon.
Vaihtoehto periytymisen käyttämiselle	Algoritmi- tai käyttäytymisjoukon voi suorittaa periyttämisen avulla esim. periyttämällä Context-luokasta (kuva 2.3) aliluokkia erilaisen käyttäytymisen toteuttamiseksi. Tällöin käyttäytyminen liitetään kiinteästi Context-luokkaan, jolloin käyttäytymisen ylläpidettävyys ja laajennettavuus kärsii. Strategia-suunnittelumallin käyttäminen periyttämisen sijaan parantaa algoritmin muunneltavuutta.
Ehtolauseet tulevat tarpeettomiksi	Ehtolausekkeiden välttäminen on vaikeaa, jos yhteen luokkaan on kasattu erilaisia käyttäytymistapoja, joista oikea pitäisi valita. Strategia-suunnittelumallissa sovellus delegoi tehtävän Strategy-olioille (kuva 2.3).
Olioiden määrä lisääntyy	Strategia-suunnittelumallin käyttö lisää olioiden määrää sovelluksessa, koska jokaista käyttäytymistapaa varten luodaan oma luokkansa.

2.5 Edustaja (Proxy)

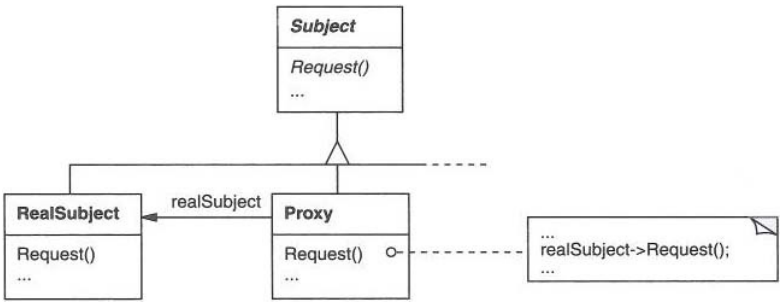
Edustaja-suunnittelumallissa asiakassovellus keskustelee komponentin edustajan kanssa, eikä suoraan komponentin kanssa (Buschmann & al., 1996). Suora yhteys komponenttiin on mahdollinen, mutta ei paras mahdollinen lähestymistapa.

Edustajaa käytetään Toivosen (2001) mukaan silloin, kun tarvitaan yksinkertaista osoitinta joustavampi tai monipuolisempi tapa viitata olioon, esimerkiksi:

- Etäedustaja (remote proxy) toimii paikallisena edustajana oliolle, joka sijaitsee toisessa osoiteavaruudessa.
- Virtuaaliedustaja (virtual proxy) luo olion vasta tarvittaessa. Olio ei siten kuluta turhaan resursseja.
- Suojausedustaja (protection proxy) kontrolloi pääsyä alkuperäiseen olioon.
- Älykäs osoitin (smart reference) suorittaa lisätoimintoja olioon kohdistuvan kutsun yhteydessä.

Proxy (kuva 2.4) pitää viitettä todelliseen kohteeseen. Proxy tarjoaa samanlaisen rajapinnan kuin Subject, jotta edustaja voi toimia todellisen subjektin korvikkeena. Proxy siirtää pyynnöt tarvittaessa RealSubject-oliolle. Proxy voi olla vastuussa kohteen luonnista ja tuhoamisesta ja se kontrolloi pääsyä todelliseen kohteeseen. Subject määrittelee rajapinnan, joka on yhteinen

RealSubject- ja Proxy-luokalle, jolloin Proxy-oliota voidaan käyttää kaikissa tilanteissa RealSubject-olion korvikkeena. RealSubject määrittelee todellisen olion, jota Proxy edustaa.



Kuva 2.4: Edustaja (Toivonen, 2001)

Taulukossa 2.5 on kuvattu mallin vaikutuksia Toivosen (2001) ja Buschmannin & al., (1996) perusteella.

Taulukko 2.5: Edustajan vaikutuksia.

Vaikutus	Selitys
Vähentää kytkentää asiakas-sovelluksen ja kutsuttavan komponentin välillä	Edustaja peittää sen, että kohde voi sijaita toisessa osoiteavaruudessa. Kutsuttavaa komponenttia (esim. liiketoimintapalveluun) voi muuttaa asiakassovellusta muuttamatta.
Tehostaa suorituskykyä	Edustaja voi suorittaa optimointia, kuten luoda olion vasta tarvittaessa ja kopioida välimuistiin luotuja olioita.
Alentaa suorituskykyä	Vaikka suunnittelumalli voi optimoida toimintaa, olioden epäsuora käyttö, sekä mahdollinen pullonkaula saattavat haitata suorituskykyä.

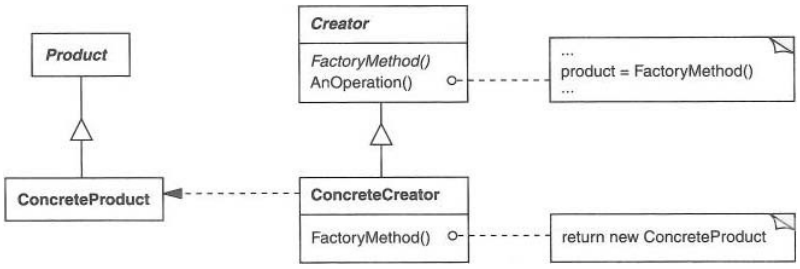
2.6 Tehdasmetodi (Factory Method)

Tehdasmetodi määrittelee olion luontioperaation kutsumuodon, mutta jättää aliluokkien tehtäväksi päättää, mistä luokasta ilmentymä luodaan (Toivonen, 2001). Tehdasmetodia käyttämällä luokka voi siirtää ilmentymien luonnin aliluokille. Metsker (2002) lisää, että asiakasohjelman ei tarvitse tietää mikä on se luokka useammasta mahdollisesta luokasta, josta ilmentymä on luotava.

Toivosen (2001) mukaan Tehdasmetodia käytetään, kun

- luokka ei ennalta tiedä, mistä luokasta sen on luotava ilmentymiä
- luokka haluaa aliluokkiensa määrittelevän luotavat oliot

- luokat delegoivat vastuun jollekin useista avustavista aliluokista ja tieto siitä, mille aliluokalle delegoidaan, halutaan paikallistaa tiettyyn kohtaan.



Kuva 2.5: Tehdasmetodi (Toivonen, 2001)

Product (kuva 2.5) määrittelee niiden olioiden rajapinnan, joita tehdasmetodi luo. ConcreteProduct toteuttaa Product-rajapinnan. Creator määrittelee tehdasmetodin, joka palauttaa Product-tyyppisen olion. Creator voi lisäksi määritellä tehdasmetodin oletustoteutuksen, joka palauttaa oletuksena olevan ConcreteProduct-olion. Creator voi kutsua tehdasmetodia Product-olion luomiseksi. Creator antaa aliluokkiensa tehtäväksi toteuttaa tehdasmetodi siten, että se palauttaa oikean ConcreteProduct-ilmentymän. ConcreteCreator korvaa tehdasmetodin metodilla, joka palauttaa ConcreteProduct-ilmentymän.

Toivosen (2001) mukaan Tehdasmetodia käyttämällä vältetään sitomasta sovellusriippuvaisia luokkia. Sen sijaan käytetään Product-rajapintaa, johon voidaan liittää mikä tahansa ConcreteProduct-luokka. Tosin asiakkaan on periytettävä Creator-luokasta uusi aliluokka jokaista ConcreteProduct-luokkaa kohden.

Taulukossa 2.6 on kuvattu mallin vaikutuksia Toivosen (2001) perusteella.

Taulukko 2.6: Tehdasmetodin vaikutuksia.

Vaikutus	Selitys
Olioiden luominen joustavaa	Olioiden luominen luokan sisällä tehdasmetodia käyttäen on joustavampaa kuin olioiden luominen suoraan. Tehdasmetodi tarjoaa aliluokille ”koukun”, johon ne voivat kiinnittää oman laajennetun versionsa oliosta.
Yhdistää rinnakkaiset luokkahierarkiat	Rinnakkainen luokkahierarkia syntyy, kun luokka delegoi osan vastuistaan toiselle erilliselle luokalle.

3 J2EE-SUUNNITTELUMALLIT

Tässä luvussa esitellään J2EE-teknologiaa hyväksikäyttäviä suunnittelumalleja. Mallit on jaettu arkkitehtonisesti kolmeen kerrokseen: esityskerros, liiketoimintakerros ja integrointikerros. Esityskerroksen malleja hyödyntäen laadittu Kurssinhallintasovellus on dokumentoitu liitteissä 1-7. Suunnittelumallien pääasiallisena lähteenä on käytetty lähdettä Alur & al. (2003), mutta kirjallisuudesta on pyritty löytämään myös muiden kokemuksia.

3.1 Esityskerroksen suunnittelumalleja

Tässä kohdassa esitellään J2EE-arkkitehtuurin esityskerroksen suunnittelumalleja. Mukana ovat kuvan 1.1 kaikki esityskerroksen suunnittelumallit. Kustakin suunnittelumallista on kerrottu kuvaus, käyttötarkoitus, vaikutukset ja toteutusstrategioita. Soveltamisesta on kerrottu niiden mallien osalta, joita on käytetty tutkielmaan liittyvässä Kurssinhallintasovelluksessa. Esimerkkisovelluksessa on käytetty suunnittelumalleja Sieppaussuodatin, Etukontrolleri, Kontekstiolio, Sovelluskontrolleri, Näkymäavustaja, Välittäjänäkymä ja Palvelu työstäjälle.

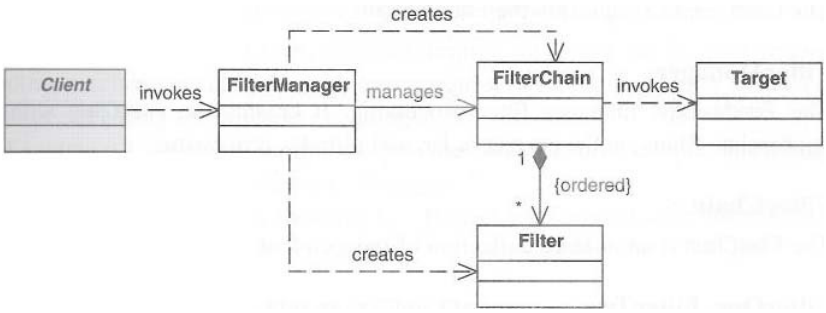
3.1.1 Sieppaussuodatin (Intercepting Filter)

Alurin & al. (2003) mukaan Sieppaussuodattimen avulla suoritetaan palvelupyynnön ja –vastauksen esi- ja jälkiprosessointi. Suodatinkäsittelijä yhdistää löyhästi kytketyt suodattimet ketjuksi ja delegoi kontrollin sopivalle suodattimelle. Ketjun suoritus käynnistyy palvelupyynnön tai –vastauksen kohdistuessa tiettyyn URL-osoitepolkuun. Suodattimia voi lisätä, poistaa ja yhdistää eri tavoin muuttamatta olemassaolevaa koodia. Suodattimia voivat olla esim. koodaustavan selvittäminen, lokiin kirjoitus, virheiden jäljitys, salaus, salauksen purkaminen tai tiivistäminen. Dragomiroiu & al. (2003) sijoittavat Sieppaussuodattimen ohjauskerrokseen MVC-mallissa. Hammouda ja Koskimies (2002) käyttävät Sieppaussuodatinta kehittämässään työkalussa, jolla luodaan sovelluksen arkkitehtoninen kehys suunnittelumallien avulla.

Alurin & al. (2003) mukaan Sieppaussuodatinta käytetään, kun halutaan saada selville

- Onko sessio validi?
- Rikkooko palvelupyynnön polku rajoituksia?
- Onko selaimen tyyppi tuettu?
- Mitä koodaustapaa asiakas käyttää?
- Onko palvelupyynnön virta (stream) salattu tai tiivistetty?

Client (kuva 3.1) lähettää palvelupyynnön FilterManagerille, joka luo FilterChain-ketjun sopivin suodattimin, oikeassa järjestyksessä ja alustaa prosessoinnin. FilterChain on järjestetty kokoelma toisistaan riippumattomia suodattimia. FilterChain koordinoi suodattimien prosessointia. Filter on yksittäinen suodatin, joka on kytketty asiakkaan palvelupyynnön haluamaan resurssiin.



Kuva 3.1: Sieppaussuodatin (Alur & al., 2003)

Taulukossa 7 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.1: Sieppaussuodattimen vaikutuksia.

Vaikutus	Selitys
Kontrolloinnin keskittäminen	Suodattimet tarjoavat keskitetyn paikan käsitellä useiden palvelupyyntöjen prosessointia. Suodattimet ”hienosäätävät” palvelupyyntöjä ennen kohderesurssin, kuten kontrollerin, käsittelyä.
Parantaa uudelleenkäytettävyyttä	Suodattimet edistävät sovelluksen uudelleenkäyttöä ja osittamista. Standardin rajapinnan avulla suodattimia voi näkymättömästi lisätä tai poistaa olemassaolevaan ohjelmistoon.
Deklaratiivinen ja joustava konfigurointi	Lukuisat palvelut voidaan yhdistää varioiden, ilman ohjelmiston ytimen uudelleenkääntämistä.
Tiedonvaihto on tehotonta	Tiedon vaihtaminen suodattimien välillä voi olla tehotonta, koska ne ovat löyhästi kytkettyinä toisiinsa. Jos suuria määriä tietoa pitää jakaa suodattimien välillä, suunnittelumalli voi tulla kalliiksi.

Sieppaussuodattimen toteutuksesta esitellään Alurin & al. (2003) kuvaama ja suosittelema strategia, Standard Filter Strategy, jota käytetään tutkielmaan liittyvässä Kurssinhallintasovelluksessa. Strategia on valittu esitettäväksi tutkielmassa, koska sitä suositellaan käytettäväksi.

Standard Filter Strategy

Suodattimet konfiguroidaan deklaratiivisesti käyttäen web-sovelluksen käyttöönottokuvaajaa (deployment descriptor), kuten servlettispesifikaatiossa on kerrottu. Spesifikaatioon liittyy

standardi mekanismi suodatinketjujen rakentamiseen ja suodattimien lisäämiseen ja poistamiseen ketjuista. Suodattimet toteuttavat standardin rajapinnan ja ne ovat löyhästi kytkettyjä toisiinsa, kuten kohderesursseihinkin, joille suodattimet suorittavat esi- tai jälkiprosessoinnin.

Soveltaminen

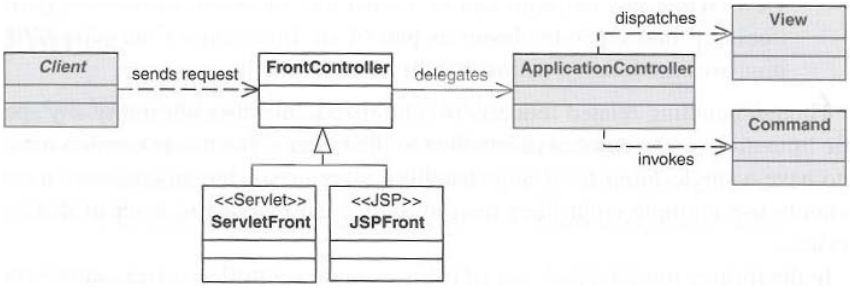
Tutkielmaan liittyvässä Kurssinhallintasovelluksessa (liite 1) Sieppaussuodatin-suunnittelumallia (liite 2) on käytetty siten, että suodattimia on kolme: BaseEncodeFilter ja siitä periytyvät StandardEncodeFilter ja MultipartEncodeFilter (liite 2). Suodattimet on asetettu käynnistymään palvelupyynnön kohdistuessa Etukontrollerille määriteltyyn osoitepolkuun. Suodattimet on ketjutettu siten, että ensin suoritetaan StandardEncodeFilter ja sitten MultipartEncodeFilter. Suodattimet jäsentävät palvelupyynnön mukana tulevat parametrit ja tallettavat ne palvelupyynnön attribuuteiksi. StandardEncodeFilter käsittelee application/x-www-form-urlencoded-koodatut parametrit eli normaalin palvelupyynnön mukana tulevat parametrit ja MultipartEncodeFilter multipart/form-data-koodatut parametrit eli multipart-lomakkeen mukana tulevat parametrit. Sovelluksen muissa osissa voidaan siten hakea palvelupyynnön parametrien arvot attribuutteina, riippumatta käytetystä koodaustavasta.

3.1.2 Etukontrolleri (Front Controller)

Etukontrolleri tarjoaa Alurin & al. (2003) mukaan keskitetyn sisäänkäynnin palvelupyyntöjen käsittelyyn. Suunnittelumalli vähentää kontrollointilogiikan kopiointia näkymiin. Esim. JSP-näkymään on houkuttelevaa upottaa suuri määrä Java-koodia, jota kutsutaan scriptlet-koodiksi. Kontrollointi kopioidaan tällöin useaan näkymään, jolloin on mahdollista, että ohjelmistoprojektin tiimit lähestyvät samaa ongelmaa eri näkökulmasta, vaikka ratkaisun tulisi olla yhdenmukainen, pakollinen ja kontrolloitu keskitetyllä tavalla. Fowlerin (2002) mukaan kontrollereita tarvitaan järjestelmässä vain yksi, joka jakaa tehtävät sopiville toiminnoille. Alurin & al. (2003) mukaan kontrollereita voi olla useita, jotka kukin palvelevat joukon erillisiä palveluita. Etukontrolleri käyttää Sovelluskontrolleria toimintojen ja näkymien hallintaan. Etukontrolleri voi käyttää Abstraktia tehdasta Kontekstiolion luomiseen web-palvelupyynnön aikana. Dragomiroiu & al. (2003) sijoittavat Etukontrollerin ohjauskerrokseen MVC-mallissa. Hammouda ja Koskimies (2002) käyttävät Etukontrolleria kehittämässään työkalussa, jolla luodaan sovelluksen arkkitehtoninen kehys suunnittelumallien avulla.

Alurin & al. (2003) mukaan Etukontrolleria käytetään, kun

- halutaan välttää kontrollointilogiikan kopioimista
- halutaan yleinen logiikka moninkertaisille palvelupyynnöille
- halutaan keskittää kontrolloitu sisäänpääsy järjestelmään.



Kuva 3.2: Etukontrolleri (Alur & al., 2003)

FrontController (kuva 3.2) on alustava palvelupyyntöjen käsittelypaikka, joka delegoi ApplicationControllerille toiminnon ja näkymän hallintaan. ApplicationControllerille etsii ja reitittää tietylle Command-oliolle, joka täyttää palvelupyynnön ja välittää sopivalle View-oliolle. Command suorittaa operaation, joka tyydyttää palvelupyynnön. View on näyttö, joka palautetaan asiakkaalle.

Taulukossa 3.2 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.2: Etukontrollerin vaikutuksia.

Vaikutus	Selitys
Kontrollin keskittäminen	Kontrolleri tarjoaa keskitetyn paikan käsitellä palvelupyyntöjen yleistä kontrollilogiikkaa
Kehittää mukautuvuutta	Kontrollin keskittäminen helpottaa laittomien sisäänpääsy-yritysten estämistä. Keskittäminen vie tässä suhteessa myös vähemmän resursseja, kuin useiden näkyminen tarkkailu.
Kehittää uudelleenkäytettävyyttä	Etukontrolleri mahdollistaa sovelluksen puhtaamman osittamisen ja komponenttien uudelleenkäytön, palvelupyyntöjen yleisen kontrolloinnin siirtyessä kontrollerin hallintaan.
Kehittää roolien erottelua	Ohjelmistoprojektin tiimien jäsenet voivat keskittyä roolilleen tyypilliseen työhön kontrollin keskittämisen myötä. Ohjelmistosuunnittelijat voivat keskittyä kontrolloinnin ohjelmointiin ja WWW-suunnittelijat ulkoasun luomiseen, kun kontrollia ei sisällytetä suoraan näyttöihin.

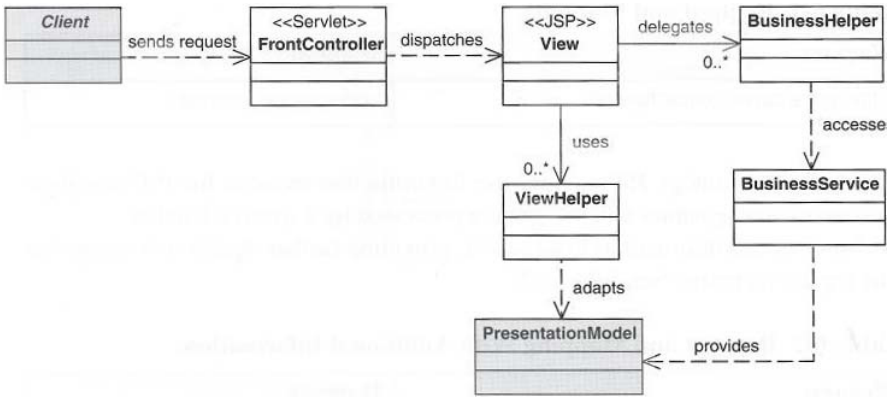
Etukontrollerin toteutuksesta esitellään kaksi Alurin & al. (2003) kuvaamaa strategiaa, Command and Controller Strategy ja Dispatcher in Controller Strategy. Tutkielmaan liittyvässä sovelluksessa on käytetty molempia strategioita. Command and Controller –strategia on valittu esitettäväksi tutkielmassa, koska se yhdistää Etukontrolleri- ja Sovelluskontrolleri – suunnitte-

lumallien toteutukset. Dispatcher in Controller –strategia on valittu esitettäväksi Command and Controller –strategian rinnalle loogisesti yhteenkuuluvana osana yksinkertaisten palvelupyyntöjen käsittelyä varten.

Command and Controller Strategy

Käytettäessä Command Handler -strategiaa Etukontrollerin kanssa, kutsutaan sitä Command and Controller -strategiaksi. Strategiassa käytetään Komento-suunnittelumallia, joka tarjoaa geneerisen rajapinnan palvelupyyntöjä palveleville avustaja-olioille. (ks. Sovelluskontrollerin Command Handler Strategy).

Dispatcher in Controller Strategy



Kuva 3.3: Dispatcher in Controller Strategy (Alur & al., 2003)

Kun näkymien hallinta ja siihen liittyvä välitystoiminnallisuus on minimaalista, molemmat voidaan hoitaa kontrollerin avulla. Strategiassa FrontController (kuva 3.3) ottaa vastaan palvelupyynnön ja välittää suoraan sopivalle näkymälle (View). Mikäli liiketoimintaprosessointia tarvitaan, se suoritetaan vasta kun kontrolli on näkymällä.

Soveltaminen

Tutkielmaan liittyvässä Kurssinhallintasovelluksessa (liite1) Etukontrolleri-suunnittelumallia on käytetty siten, että yksinkertaisissa näkymissä välitystoiminta hoidetaan kontrollerin (liite 6) avulla (Dispatcher in Controller Strategy). Jos palvelupyynnön käsittelyyn sisältyy toiminnallisuutta, esim. tietokantahakuja, kontrolli delegoidaan Sovelluskontrolleri-suunnittelumal-

lia hyväksikäyttävälle KurssiSovellusKontrolleri-luokalle (liite 5) (Command and Controller Strategy).

3.1.3 Sovelluskontrolleri (Application Controller)

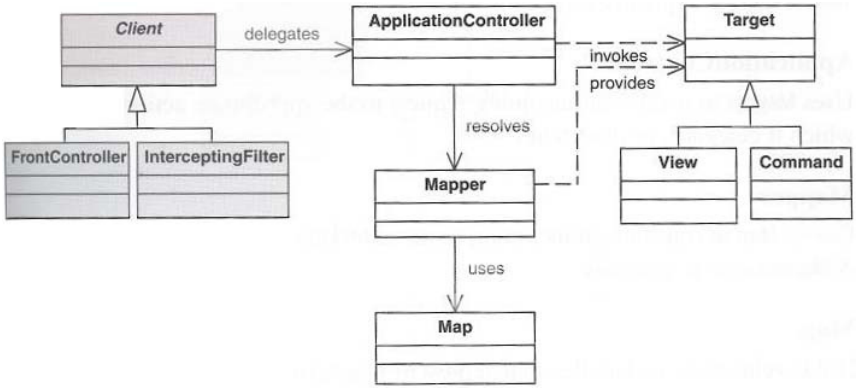
Sovelluskontrolleri hallinnoi toimintoja ja näkymiä. Alurin & al. (2003) mukaan toiminnon hallinnan avulla etsitään ja kutsutaan sopivia toimintoja, joiden avulla käsitellään palvelupyynnöt. Toimintoja ovat esim. validointi, autentikointi, pääsyn kontrollointi, jne. Näkymien hallinnan avulla välitetään suoritus sopivaan näkymään tai näkymän generointimekanismiin. Knight ja Dai (2002) käyttävät erityistä, toimintaperiaatteiltaan Alurin & al. suunnittelumallia muistuttavaa Sovelluskontrolleria erottamaan sovellusvuon liiketoimintalogiikasta MVC-mallia käytettäessä.

Toimintojen ja näkymien hallinnan erottaminen protokollariippuvaisesta Etukontrollerista on hyödyllistä. Protokollariippumattomuus saavutetaan Kontekstiolion avulla, jota käytetään Etukontrollerin ja Sovelluskontrollerin väliseen kommunikointiin. Joitakin Sovelluskontrollerin komponentteja voidaan uudelleenkäyttää käsittelemään palvelupyyntöjä useammasta kanavasta, kuten web-sovelluksesta tai web-palvelusta. Fowlerin (2002) mukaan Sovelluskontrolleri on erityisen hyödyllinen silloin kun käyttäjälle täytyy näyttää tietyt näytöt tietyssä järjestyksessä ja näkymät ovat riippuvaisia olioiden tiloista.

Alurin & al. (2003) mukaan Sovelluskontrolleria käytetään, kun

- halutaan uudelleenkäyttää toimintojen ja näkymien hallinnan koodia
- halutaan kehittää palvelupyyntöjen käsittelyn laajennettavuutta, lisäämällä käyttötapauksien toiminnallisuutta inkrementaalisesti
- halutaan kehittää koodin modulaarisuutta ja ylläpidettävyyttä

Client (kuva 3.4) on asiakas, joka kutsuu ApplicationControlleria. Client voi olla Etukontrolleri tai Sieppaussuodatin. ApplicationController käyttää Mapperia välittääkseen palvelupyynnön sopivalle toiminnolle ja näkymälle. Mapper käyttää Mapia kääntääkseen palvelupyynnön sopivalle toiminnolle ja näkymälle. Map säilyttää viitteet kahvoille, jotka ovat kohderesursseja (Target). Mapit voidaan toteuttaa luokkina tai rekistereinä. Target on resurssi, joka auttaa täyttämään palvelupyynnön. Resursseja ovat komennot ja näkymät.



Kuva 3.4: Sovelluskontrolleri (Alur & al., 2003)

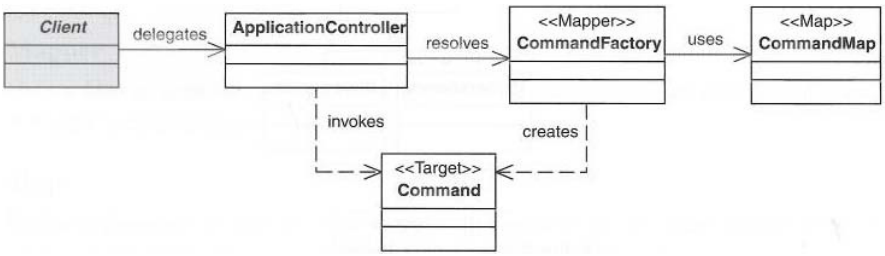
Taulukossa 3.3 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.3: Sovelluskontrollerin vaikutuksia.

Vaikutus	Selitys
Kehittää modulaarisuutta	Modulaarisuus ja uudelleenkäytettävyys paranee, kun toimintojen ja näkymien hallinnointi erotetaan protokollariippuvaisesta Etukontrollerista. Sovelluksen laajentaminen ja testaaminen helpottuu palvelupyyntöjen ollessa riippumattomia web-säiliöstä.
Kehittää uudelleenkäytettävyyttä	Yleisiä ja modulaarisia komponentteja voi uudelleenkäyttää.
Kehittää laajennettavuutta	Toiminnallisuutta voidaan lisätä protokollariippumattomasti ja deklaratiivisesti, jolloin koko ohjelmiston koodia ei tarvitse kääntää uudelleen.

Sovelluskontrollerin toteutuksesta esitellään kaksi Alurin & al. (2003) kuvaamaa strategiaa, Command Handler Strategy ja View Handler Strategy. Strategioita käytetään usein yhdessä, kuten tehdään tutkielmaan liittyvässä sovelluksessa. Strategiat on valittu esitettäväksi tutkielmassa, koska ne yhdistävät Etukontrolleri- ja Sovelluskontrolleri –suunnittelumallien toteutukset.

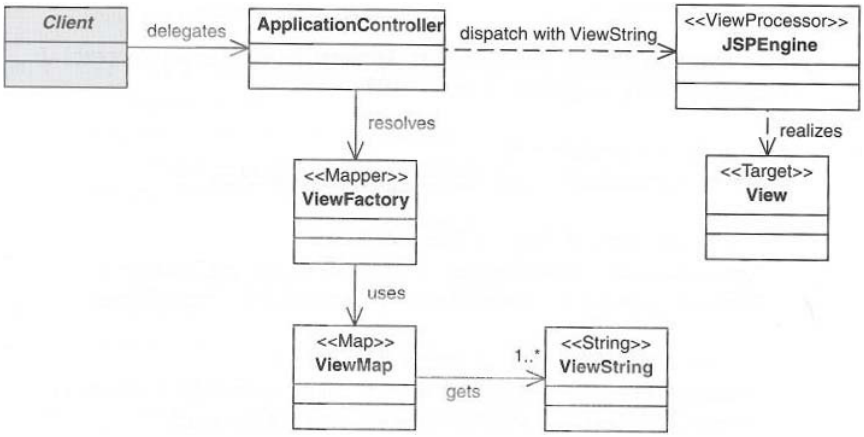
Command Handler Strategy



Kuva 3.5: Command Handler Strategy (Alur & al., 2003)

Strategiassa (kuva 3.5) Sovelluskontrolleri toimii komentojen käsittelijänä (Command Handler), joka hakee ja kutsuu Komento-suunnittelumallin mukaisia komento-olioita (Command). Komentojen käsittelijä hallitsee komentojen elinkaarta. Yhdessä palvelupyynnössä käsitellään useita käyttötapauksia. Komentoja luodaan Tehdasmetodi-suunnittelumallin mukaisesti, jolloin komennot ovat löyhästi kytkettyjä niitä kutsuviin olioihin.

View Handler Strategy



Kuva 3.6: View Handler Strategy (Alur & al., 2003)

Strategiassa (kuva 3.6) Sovelluskontrolleri toimii näyttöjen käsittelijänä (View Handler), joka hakee ja välittää sopivalle näytölle. Command Handler ja View Handler -strategiat toimivat samalla periaatteella ja niitä käytetään yhdessä, vaikka ovat toisistaan loogisesti poikkeavia toimintoja. Kuvassa 3.6 ApplicationController delegoi ViewFactorylle, joka palauttaa ViewString-kahvan ApplicationControllerille. ApplicationController välittää kahvan avulla oikealle JSP-näytölle, joka toteuttaa näkymän (View).

Soveltaminen

Tutkielmaan liittyvässä Kurssinhallintasovelluksessa (liite 1) Sovelluskontrolleri-suunnittelumallia on käytetty siten, että asiakasta edustava EtuKontrolleri (liite 6) delegoi kontrollin KurssiSovellusKontrolleri-luokalle (liite 5), joka hakee Tehdasmetodi-suunnittelumallia hyväksikäyttävältä KomentoTehdas-luokalta Komento-suunnittelumallin mukaisia komentoja, jotka suorittavat palvelupyynnön mukaisen operaation (Command Handler Strategy). Operaatio esim. hakee tarkoitusta palvelevan apuluokan avulla tietokannasta tietoja ja palaut-

taa sopivan näkymän osoitteen. Komennon suorittamisessa on siten yhdistetty operaation suorittaminen ja näkymän (liite 3) hakeminen. Sovelluskontrolleri ohjaa suorituksen JSP-näkymälle (View Handler Strategy). KomentoTehdas on toteutettu yksinkertaistamisen vuoksi siten, että olemassaolevat operaatiot asetetaan koodissa luokan HashMap-olioon. Hyvän suunnittelutavan mukaan operaatiot pitäisi haekea dynaamisesti esimerkiksi XML-tiedostosta.

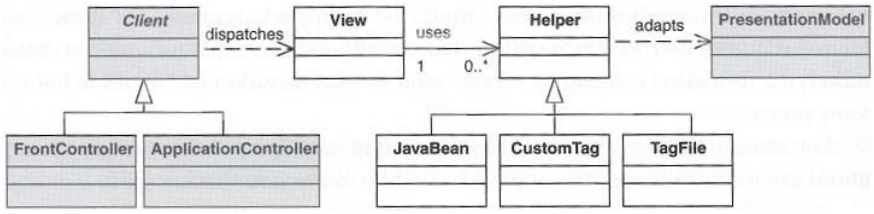
3.1.4 Näkymäavustaja (View Helper)

Näkymäavustajan avulla erotetaan Alurin & al. (2003) mukaan näkymän muotoilu ja näkymän prosessointilogiikka toisistaan. Näkymä delegoi prosessointilogiikan avustajille, jotka toimivat adaptereina liiketoimintamallin ja näkymän välillä. Avustaja prosessoi muotoilulogiikan, esim. generoi HTML-taulukon. Prosessointilogiikan erottaminen näkymästä tekee sovelluksesta modulaarisemman ja mahdollistaa komponenttien uudelleenkäytön. Suunnittelumalli soveltuu parhaiten käytettäväksi mallipohjaisten näkymien, kuten JSP, kanssa. Dragomiroiu & al. (2003) sijoittavat Näkymäavustajan näkymäkerrokseen MVC-mallissa. Hammouda ja Koskimies (2002) käyttävät Näkymäavustajaa kehittämässään työkalussa, jolla luodaan sovelluksen arkkitehtoninen kehys suunnittelumallien avulla

Suunnittelumallin tavoitteena on osittaa prosessointilogiikka, kuten kontrollointi-, muotoilu-, liiketoiminta- ja tiedonhakulogiikka näkymän ulkopuolelle. Liiketoimintalogiikka on tavallisesti oliomallissa, kuten Kuljetusoliossa tai Liiketoimintaoliossa. Tiedonhakulogiikka on kapseloitu Tiedonhakuoliossa. Kontrollointilogiikka on Etukontrollerissa, komentojen, avustajien ja avustajien muotoilulogiikan kanssa. Näkymäavustajan näkymä kapseloi näkymän muotoilun ja avustaja kapseloi näkymän prosessointilogiikan. Avustaja voi olla papu, kustomoitu tagi tai tagitiedosto.

Alurin & al. (2003) mukaan Näkymäavustajaa käytetään, kun

- halutaan käyttää mallipohjaisija näkymiä, kuten JSP
- halutaan välttää ohjelmointilogiikan upottamista näkymään
- halutaan erottaa ohjelmointilogiikka näkymästä



Kuva 3.7: Näkymäavustaja (Alur & al., 2003)

Client (kuva 3.7) on asiakas (FrontController, ApplicationController), joka välittää suorituksen näkymään. View (näkymä) näyttää tiedon asiakkaalle. Esitysmallin dynaaminen tieto haetaan ja konvertoidaan avustajien avulla. Helper (avustaja) kapseloi näkymän generoinnin ja muotoilun prosessointilogiikan. Avustaja muuntaa esitysmallin (PresentationModel) näkymälle tai tarjoaa pääsyn esitysmallin ”raakaan” tietoon. PresentationModel säilyttää liiketoimintapalveluilta haettua tietoa, jota käytetään näkymän (View) generointiin.

Taulukossa 3.4 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.4: Näkymäavustajan vaikutuksia.

Vaikutus	Selitys
Parantaa sovelluksen osittamista, uudelleenkäyttöä ja ylläpidettävyyttä	WWW-sivujen tekemisen ja prosessointilogiikan, kuten kontrollointi-, muotoilu-, liiketoiminta- ja tiedonhakulogiikan erottaminen johtaa jalostettuun sovelluksen osittamiseen. Toisiinsa loogisesti liittymättömät sovelluksen osat voidaan kapseloida koossapitäviin ja uudelleenkäytettäviin komponentteihin. Kontrollointilogiikka voidaan siirtää Etukontrolleriin ja avustajiin, liiketoimintalogiikka on Liiketoimintaoliossa ja tiedonhakulogiikka on Tiedonhakuoliossa.
Parantaa roolien erottelua	Ohjelmiston kehittäjät ja WWW-suunnittelijat voivat työskennellä samanaikaisesti toisiinsa liittyvien sovelluksen osien parissa, koska prosessointilogiikka on näkymän ulkopuolella.
Helpottaa testausta	Prosessointilogiikan osittaminen useampaan avustaja-komponenttiin, helpottaa testausta. Prosessointilogiikan sisältävän JSP-sivun testaaminen on vaikempaa kuin useamman erillisen luokan testaaminen.
Avustajien käyttö saattaa peilata scriptlettien käyttöä	Prosessointilogiikan eräs hyöty on vähentää koodin upottamista suoraan näytölle. Jos esim. JSP-sivulla viitataan avustajiin konditionaalisilla ehdoilla, alkaa sivu näyttää ohjelmointilogiikan upottamiselta näytölle.

Näkymäavustajan toteutuksesta esitellään neljä Alurin & al. (2003) kuvaamaa strategiaa; yhtä aikaa käytettävät Template-Based View Strategy ja Javabean Helper Strategy, sekä tutkielmaan liittyvässä sovelluksessa käytettävät Custom Tag Helper Strategy ja Tag File Helper Strategy. Template-Based View- ja Javabean Helper –strategiat on valittu esitettäväksi tutkielmassa, koska ne kuvaavat suunnittelumallien toteuttamista EJB-teknologiaa käyttäen. Custom Tag Helper- ja Tag File Helper –strategiat ovat vaihtoehtoisia tapoja Java-pavuille käyttää hyväksi Näkymäavustaja-suunnittelumallia.

Template-Based View Strategy & JavaBean Helper Strategy

Template-Based View ja JavaBean Helper -strategioita käytetään yhtä aikaa. Strategiassa käytetään mallipohjaista näkymää, kuten JSP:tä. Mallipohja näyttää avustaja-pavun sisältämien ominaisuuksien arvoja käyttäjälle palautettavalla WWW-sivulla. Etukontrolleri asettaa JavaBean-olion (welcomeHelper) palvelupyynnön alueelle ja välittää JSP-näkymälle. Kuvassa (kuva 3.8) on esimerkki mallipohjaisesta näkymästä.

```
<jsp:useBean id="welcomeHelper" scope="request"
 class="corepatterns.util.WelcomeHelper" />
<HTML>
<BODY bgcolor="FFFFFF">
<c:if test = "${welcomeHelper.nameExists == true}">
<center><H3> Welcome <b>
<c:out value='${welcomeHelper.name}'/>
</b><br><br> </H3></center>
</c:if>
<H4><center>Glad you are visiting our site!</center></H4>
</BODY>
</HTML>
```

Kuva 3.8: Mallipohjainen näkymä (Alur & al., 2003)

Custom Tag Helper Strategy

Strategiassa avustaja toteutetaan kustomoidun tagin avulla. Kustomoitu tagi sovittaa mallin näytölle sopivaksi. Strategian hyöty on kustomoidun tagin joustavuudessa ja laajennettavuudessa, vaikka se vaatiikin enemmän suunnittelu- ja toteutustyötä kuin esimerkiksi JavaBean Helper Strategy. Kustomoitu tagi tehdään tagin (java-luokka), tagi-kirjaston kuvauksen ja asetustiedoston avulla.

Tag File Helper Strategy

Strategiassa avustaja toteutetaan tagitiedoston avulla. Tagitiedostoon kirjoitetaan uudelleenkäytettävä koodi, jota kutsutaan JSP-näkymästä. Tagitiedoston avulla voidaan esimerkiksi luoda HTML-taulukko tietokannasta haetuille tiedoille. Tagitiedostoa voidaan kutsua mistä tahansa JSP-näkymästä ja välittää sopivat muotoilut parametreina. JSP-tiedostoon ei tällöin tarvitse kirjoittaa silmukkarakenteita.

Soveltaminen

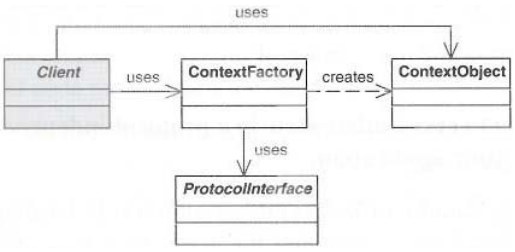
Tutkielmaan liittyvässä Kurssinhallintasovelluksessa (liite 1) Näkymäavustaja-suunnittelumallia on käytetty Custom Tag Helper ja Tag File Helper -strategioiden tapaan. Vertailun vuoksi sama toiminnallisuus on kirjoitettu sekä kustomoidun tagin (TilakuvausTag.java) että tagitiedoston (tilakuvaus.tag) avulla (liite 3). Lisäksi toisen kustomoidun tagin (KurssiListaTag.java) avulla muodostetaan taulukko tietokannasta saadulle tulosjoukolle. Monimutkaisempaa tagitiedostoa edustaa tiedostoupload.tag, jonka avulla kutsutaan kontrollerin kautta toimintoa, joka lähettää tiedoston palvelimelle. Saman tagitiedoston avulla lähetetty (ja tallennettu) tiedosto voidaan ladata palvelimelta. Tagitiedostoa, kuten kustomoitua tagiakin, voidaan käyttää uudelleen useassa JSP-näkymässä.

3.1.5 Kontekstiolio (Context Object)

Kontekstiolion avulla Alurin & al. (2003) mukaan kapseloidaan järjestelmän tietoa protokollasta riippumatta, jolloin se voidaan jakaa sovelluksen muihin komponentteihin kytkemättä niitä tiettyyn protokollaan. Esim. HTML-lomakkeen tiedot voidaan tallentaa Kontekstiolioon. Muut sovelluksen osat pääsevät lomakkeen sisältämään tietoon käsiksi Kontekstiolion kautta tietämättä HTTP-protokollasta. Muutokset protokollaan hoidetaan Kontekstiolion kautta, muita sovelluksen osia muuttamatta.

Alurin & al. (2003) mukaan Kontekstioliota käytetään, kun

- järjestelmässä on komponentteja ja palveluita, jotka tarvitsevät järjestelmän tietoja
- halutaan irtikytkeä sovelluksen komponentit ja palvelut protokollaspesifistä järjestelmätiedoista
- halutaan paljastaa vain relevantti API kontekstin sisällä.



Kuva 3.9: Kontekstiolio (Alur & al., 2003)

Client (kuva 3.9) luo olion ProtocolInterfacen avulla. ProtocolInterface paljastaa protokollan tai kerros-spesifejä yksityiskohtia. ContextFactory luo protokolla- ja kerrosriippumattoman ContextObjectin. ContextObject on geneerinen olio, jota käytetään palvelinriippumattoman tilan jakamiseen sovelluksessa.

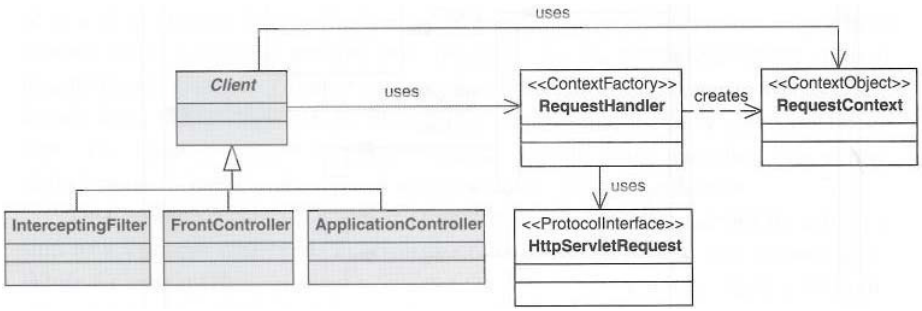
Taulukossa 3.5 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.5: Kontekstiolion vaikutuksia.

Vaikutus	Selitys
Parantaa uudelleenkäytettävyyttä ja ylläpidettävyyttä	Sovelluksen komponentit ja alijärjestelmät ovat geneerisempiä ja niitä voidaan uudelleenkäyttää eri tyyppisien asiakassovelluksien kanssa, koska sovelluksen rajapintojen ei tarvitse huolehtia protokolla-spesifeistä tietotyypeistä.
Parantaa testattavuutta	Helpottaa poistamaan riippuvuudet protokolla-spesifistä koodista, joka voi sitoa ajonaikaisen ympäristön tiettyyn säiliöön, esimerkiksi www-palvelimeen tai sovelluspalvelimeen. Testaaminen on helpompaa riippuvuuksien poiston jälkeen, koska esim. automaattiset testausvälineet toimivat suoraan Kontekstiolioiden kanssa.
Vähentää rajapintojen kehityksen rajoituksia	Kontekstiolio kapseloi lukuisia olioita. Rajapinnat ovat vähemmän riippuvaisia yksityiskohdista, kun ne hyväksyvät Kontekstiolion, eikä kaikkia sen sisältämiä olioita.
Alentaa suorituskykyä	Mallin käyttö aiheuttaa pienen suorituskyvyn laskun, mikä johtuu tilan kuljettamisesta oliolta toiselle.

Näkymäavustajan toteutuksesta esitellään Alurin & al. (2003) kuvaama strategia, Request Context POJO Strategy. Strategiaa käytetään tutkielmaan liittyvässä Kurssinhallintasovelluksessa, koska se parhaiten soveltuu toteutettavaksi kyseisessä järjestelmässä yksinkertaisuuden vuoksi. Muut kirjallisuudessa esitetyt strategiat koskevat esimerkiksi Struts-kehystä, au-ten-ti-koin-tia tai HTML-validointia, joita sovelluksessa ei käytetä.

Request Context POJO Strategy



Kuva 3.10: Request Context POJO Strategy

Strategiassa (kuva 3.10) Kontekstiolio toteutetaan tavallisena java-oliona, joka luodaan Tehdasmetodin mukaisesti. Kontekstikäsittelijä (RequestHandler) luo Kontekstiolion (RequestContext), jolle se siirtää HttpServletRequest-palvelupyynnön tilan. Asiakas voi olla Sieppaussuodatin, Etukontrolleri tai Sovelluskontrolleri.

Soveltaminen

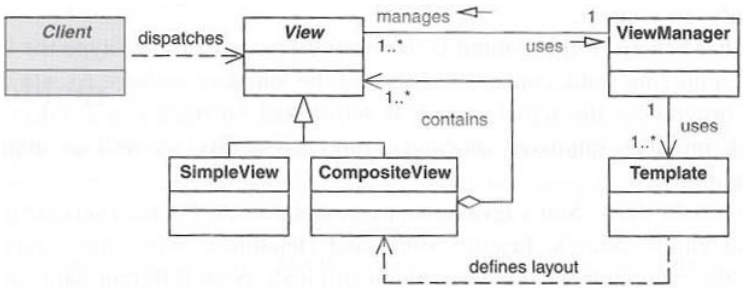
Tutkielmaan liittyvässä Kurssinhallintasovelluksessa (liite 1) Kontekstiolio-suunnittelumallia on käytetty Request Context POJO -strategian mukaisesti. Kontekstioliot (liite 4) ovat tavallisia Java-olioita (Kurssi.java, Opiskelija.java ja Opettaja.java), joita luodaan tarvittaessa KontekstiOlioTehdas-luokan avulla Tehdasmetodi-suunnittelumallin mukaisesti.

3.1.6 Komposiittinäkymä (Composite View)

Komposiittinäkymä on Alurin & al. (2003) mukaan koostettu useista atomisista alinäkymistä. Jokainen alinäkymä voidaan lisätä dynaamisesti kokonaisuuteen ja sivun ulkoasua voidaan hallita ja muokata riippumatta alinäkymien sisällöstä. Komposiittinäkymän avulla voidaan generoida WWW-sivuja, joiden sisältämiä komponentteja voidaan yhdistellä monin eri tavoin. Komponentteja eli alinäkymiä voivat olla esim. uutistiedotteet, säätiedot ja pörssitiedotteet, joita voidaan käyttää samalla sivulla. WWW-suunnittelijat voivat suunnitella sivujen ulkoasua asettamalla alinäkymiin staattista tietoa. Staattinen tieto korvataan myöhemmin varsinaisella dynaamisella sisällöllä. Silva ja Moreira (2005) käyttävät MVC-mallissa Komposiittinäkymää näkymän toteutuksessa.

Alurin & al. (2003) mukaan Komposiittinäkymää käytetään, kun

- halutaan uudelleenkäyttää yleisiä alinäkymiä, kuten ylätunnisteita, alatunnisteita ja taulukoita useissa näkymissä ja eri kohdissa sivun ulkoasua
- alinäkymien sisältö vaihtuu usein tai niihin liittyy kontrolleja, kuten rajoitettu käyttöoikeus tietyille käyttäjärooleille
- halutaan välttää alinäkymien upottamista ja kopiointia suoraan useisiin näkymiin, mikä hankaloittaisi ulkoasun muuttamisen hallintaa ja ylläpitoa.



Kuva 3.11: Komposiittinäkymä (Alur & al., 2003)

Client (kuva 3.11) on asiakassovellus joka välittää View-oliolle. View on näyttö. SimpleView on atominen osa kokonaisuudesta, alinäkymä. CompositeView on koostettu useista View-olioista. Jokainen View on joko SimpleView tai CompositeView. Template on näkymän ulkoasu. ViewManager käyttää Templatea asettaakseen ulkoasun, johon sisältö sijoitetaan. ViewManager käsittelee sisältöä ja ulkoasua toisistaan riippumattomasti.

Taulukossa 3.6 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.6: Komposiittinäkymän vaikutuksia.

Vaikutus	Selitys
Parantaa modulaarisuutta ja uudelleenkäyttöä	Atomisia mallin osia, kuten esimerkiksi taulukkomaisia pörssinoteerauksia, voidaan uudelleenkäyttää lukuisissa näkymissä.
Lisää rooli- tai oikeusperustaisen kontrolloinnin	Alinäkymiin voidaan liittää ajonaikainen ehto, esimerkiksi käyttöoikeus tietylle roolille.
Parantaa ylläpidettävyyttä	Muutoksien hallinta mallin (template) osissa on tehokkaampaa, kun malli ei ole kiinteästi ohjelmoitu näyttöön. Mallin sisällön osia voi muokata riippumatta ulkoasusta, kun sisältö on erotettu näkymästä.
Vähentää ylläpidettävyyttä	Sivujen muodostaminen atomisista osista on virhealtista. Esim. jos jokainen alinäkymä tuottaa HTML-sivun alku- ja lopetustagin, sivun HTML-koodi ei ole validia.
Alentaa suorituskykyä	Näytön generoiminen lukuisista alinäkymistä voi hidastaa suorituskykyä. Alinäkymien ajonaikainen koostaminen WWW-sivuksi aiheuttaa viiveen aina kun sivu tarjotaan asiakkaalle. Vaihtoehtoinen tapa on koostaa sivu alinäkymistä käännösvaiheessa, mutta tämä rajoittaa alinäkymän muuttumisen vain uudelleenkäännöksessä.

Komposiittinäkymän toteutuksesta esitellään Alurin & al. (2003) kuvaama ja suosittelema strategia, Custom Tag View Management Strategy. Strategia on valittu esitettäväksi tutkielmassa, koska sitä on suositeltu kirjallisuudessa.

Custom Tag View Management Strategy

Strategiassa näkymien hallinta suorittaa sisällön ja ulkoasun hallinnointia kustomoitujen tagien avulla. Tageja ovat: section, region ja template. Section-tagi on uudelleenkäytettävä komponentti, joka renderöi HTML- tai JSP-sivun. Region kuvaa sisällön määrittelemällä section-tageja. Template on malli, joka kontrolloi region- ja section-tagien ulkoasua renderöidyllä sivulla. Region-tagi toimii näkymien hallinnoijana, joka käyttää mallia (template) tietyn ulkoasun luomiseksi ja populoi mallin loogiset osat (section) sopivalla sisällöllä. Kuvissa 3.12-3.14 on esimerkki region- ja section-tagien määrittelystä, mallista (portal.jsp) ja section-alinäkymästä (banner.jsp).

```
<region:render template='portal.jsp'>
                    <region:put section='banner'       content='banner.jsp' />
                    <region:put section='controlpanel' content='ProfilePane.jsp' />
                    <region:put section='mainpanel'    content='mainpanel.jsp' />
                    <region:put section='footer'       content='footer.jsp' />
</region:render>
```

Kuva 3.12: Region- ja section-tagien määrittely (Alur & al., 2003)

```
<region:render section='banner'/>
<table width="100%">
   <tr align="left" valign="middle">
      <td width="20%">
         <!-- menu region -->
         <region:render section='controlpanel' />
      </td>
      <td width="70%" align="center">
         <!-- contents -->
         <region:render section='mainpanel' />
      </td>
   </tr>
</table>
</region:render>
```

Kuva 3.13: Template – portal.jsp (Alur & al., 2003)

```
<table width="100%" bgcolor="#C0C0C0">
<tr align="left" valign="middle">
                <td width="100%">
                <table align="left" border=1 width="100%">
                <tr align="left" valign="middle">
                                <td>Logo</td>
                                <td><center>Sun Java Center</td>
                </tr>
                </table>
                </td>
</tr>
</table>
```

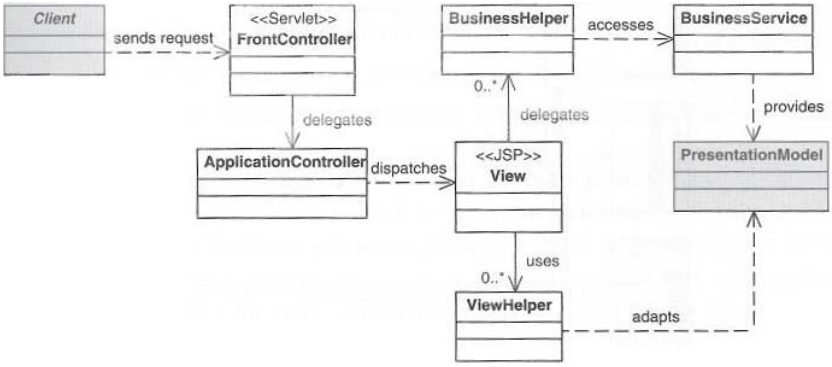
Kuva 3.14: Section-alinäkymä - banner.jsp (Alur & al., 2003)

3.1.7 Välittäjänäkymä (Dispatcher View)

Välittäjänäkymä Alurin & al. (2003) mukaan vastaa palvelupyyntöön silloin, kun vastauksen generoimiseen tarvitaan vain vähän tai ei ollenkaan dynaamista sisältöä. Vastaus voi olla kokonaan staattinen tai dynaaminen, mutta se on generoitu olemassaolevasta esitysmallista. Olemassaoleva esitysmalli on tallennettuna tilapäiseen varastoon, kuten HttpSession-olioon. Mikäli liiketoimintaprosessointia tarvitaan, se suoritetaan vasta kun kontrolli on näkymällä. Jos palvelupyynnön vastaukseen tarvitaan suurta dynaamista komponenttia ja esitysmalli haetaan liiketoimintapalvelukutsun avulla, käytetään Välittäjänäkymän sijasta Palvelu työstäjälle – suunnittelumallia. Välittäjänäkymää ja Palvelu työstäjälle -suunnittelumallia voidaan käyttää rinnakkain samassa järjestelmässä eri käyttötapauksissa. Sauter & al. (2004) laajentavat MVC-mallia Välittäjänäkymä-suunnittelumallin ja tehtäväperustaisen siirtymämallin avulla. Laajennuksen avulla voidaan paremmin käsitellä erilaisten päätelaitteiden toisistaan poikkeavia näkymiä ja siirtymiä näyttöjen välillä. Hammouda ja Koskimies (2002) käyttävät Välittäjänäkymää kehittämässään työkalussa, jolla luodaan sovelluksen arkkitehtoninen kehys suunnittelumallien avulla

Alurin & al. (2003) mukaan Välittäjänäkymää käytetään, kun

- järjestelmässä on staattisia näkymiä
- näkymiä generoidaan olemassaolevasta esitysmallista
- järjestelmässä on näkymiä, jotka ovat riippumattomia liiketoimintapalveluista
- järjestelmässä on rajoitettua liiketoimintaprosessointia.



Kuva 3.15: Välittäjänäkymä (Alur & al., 2003)

FrontController (kuva 3.15) käsittelee alustavasti palvelupyynnöt, mutta sen vastuuta on rajoitettu. ApplicationController suorittaa rajoitetusti näkymien hallintaa, mutta ei toimintojen hallintaa. Web-säiliö voi täyttää sovelluskontrollerin roolit ja vastuut. View esittää ja näyttää tiedon asiakkaalle. Esitysmalli on muokattu näyttämistä varten Näkymäavustajan avulla. Näkymä voi olla Komposiittinäkymä.

ViewHelper hakee ja muokkaa esitysmallin (PresentationModel) avustaakseen näkymän luomisessa. BusinessHelper avustaa kontrolleria alustamaan liiketoimintaprosessin käsittelemään palvelupyyntöä.

PresentationModel säilyttää liiketoimintapalvelulta haettua tietoa, jota käytetään näkymän luomisessa. BusinessService kapseloi liiketoimintalogiikan ja –tilan. Liketoimintapalveluun päästään Liiketoimintadelegaatin kautta.

Taulukossa 3.7 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.7: Välittäjänäkymän vaikutuksia.

Vaikutus	Selitys
Sovelluskehyksien ja kirjastojen hyödyntäminen	Kehykset ja kirjastot realisoivat ja tukevat tiettyjä suunnittelumalleja. Välittäjänäkymä on tuettu standardeissa ja kustomoiduissa kirjastoissa, jotka tarjoavat näkymäsovittimet, muuntajat ja rajoitettuun käyttöön tiedonhakutagit (data access tags). Esim. JSTL (Java Server Pages Standard Tag Library) on standardi kirjasto.
Voi johtaa heikkoon erotteluun kontrollin logiikan ja mallin näkymän välillä	Välittäjänäkymässä liiketoimintaprosessointi voidaan suorittaa näkymässä. Liiketoimintaprosessoinnin upottaminen tulee minimoida ja pyrkiä erottelemaan liiketoimintalogiikka näkymistä.
Erottelee prosessointilogiikan näkymistä ja kehittää uudelleenkäytettävyyttä	Näkymäavustaja muokkaa ja muuntaa esitysmallin näkymälle. Prosessointilogiikka sijoitetaan avustajiin, ei näkymiin.

Välittäjänäkymää voidaan käyttää strategian Dispatcher in Controller Strategy (ks. Etukontrolleri) mukaisesti. Tutkielmaan liittyvässä Kurssinhallintasovelluksessa suunnittelumallia käytetään tämän strategian mukaisesti (liite 6).

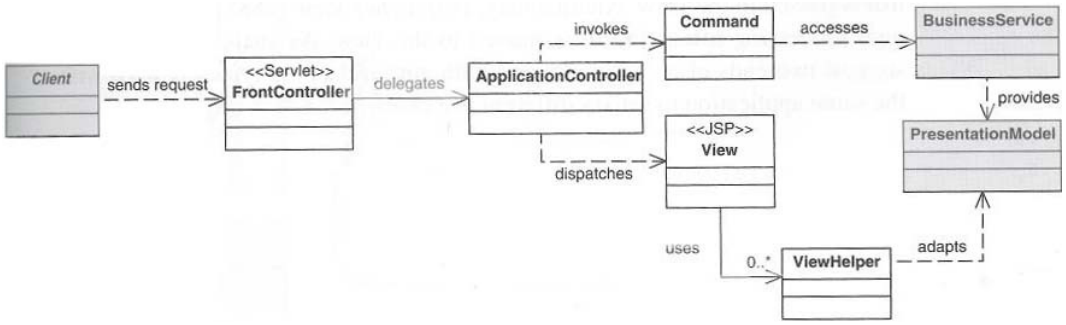
3.1.8 Palvelu työstäjälle (Service to Worker)

Palvelu työstäjälle -suunnittelumallin avulla Alurin & al. (2003) mukaan palvelupyynnön saapuessa esitysmalli muodostetaan ennenkuin kontrolli palautetaan näkymälle. Näkymä generoi dynaamisen vastauksen perustuen esitysmalliin. Palvelu työstäjälle -suunnittelumalli koostuu useista muista malleista. Keskitetty kontrolli, palvelupyyntöjen käsittely ja näkymän luominen tehdään Etukontrollerin, Sovelluskontrollerin ja Näkymäavustajan avulla. Palvelu työstäjälle -suunnittelumalli on kontrollikeskeinen arkkitehtuuri, kun taas Välittäjänäkymä on näkymäkeskeinen arkkitehtuuri. Sauter & al. (2004) laajentavat MVC-mallia Palvelu työstäjälle-suunnittelumallin ja tehtäväperustaisen siirtymämallin avulla. Laajennuksen avulla voidaan paremmin käsitellä erilaisten päätelaitteiden toisistaan poikkeavia näkymiä ja siirtymiä näyttöjen välillä.

Alurin & al. (2003) mukaan Palvelu työstäjälle -suunnittelumallia käytetään, kun

- palvelupyynnön saapuessa, jokin tietty liiketoimintalogiikka suoritetaan, jotta saataisiin dynaamisen vastauksen muodostamiseen tarvittava sisältö
- käytettävä näkymä riippuu liiketoimintapalvelun vastauksesta.

FrontController (kuva 3.16) käsittelee palvelupyynnöt alustavasti ja delegoi ne sovelluskontrollerille toiminnan ja näkymän hallintaan. ApplicationController on vastuussa toiminnan ja näkymän hallinnasta. Se valitsee palvelupyyntöön sopivan toiminnan ja näkymän.



Kuva 3.16: Palvelu työstäjälle (Alur & al., 2003)

View esittää ja näyttää tiedon asiakkaalle. Esitysmalli on muokattu näyttämistä varten Näkymäavustajan avulla. Näkymä voi olla Komposiittinäkymä. ViewHelper avustaa näkymää tai kontrolleria suorittamaan tietyn prosessoinnin. ViewHelper hakee ja muokkaa esitysmallin (PresentationModel) avustaakseen näkymän luomisessa. BusinessHelper avustaa kontrolleria alustamaan liiketoimintaprosessin käsittelemään palvelupyyntöä.

PresentationModel säilyttää liiketoimintapalvelulta haettua tietoa, jota käytetään näkymän luomisessa. BusinessService kapseloi liiketoimintalogiikan ja –tilan. Liiketoimintapalveluun päästään Liiketoimintadelegaatin kautta.

Taulukossa 3.8 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella:

Taulukko 3.8: Palvelu työstäjälle vaikutuksia.

Vaikutus	Selitys
Keskittää kontrollin ja kehittää modulaarisuutta, uudelleenkäytettävyyttä ja ylläpidettävyyttä	Yleistä palvelupyynnön prosessointia voidaan uudelleenkäyttää, jolloin vältetään prosessointilogiikan koodin kopiointia näkymiin. Muutokset tehdään vain keskitettyyn paikkaan, eikä jokaiseen näyttöön.
Kehittää roolien erottelua	Kontrollin ja palvelupyynnön prosessoinnin keskittäminen erottaa sen näkymien luomisesta. WWW-sivujen tekijät voivat keskittyä ulkoasun tekemiseen ja ohjelmiston kehittäjät ohjelmointilogiikkaan.

Seuraavia strategioita voidaan käyttää Palvelu työstäjälle -suunnittelumallissa:

- Command and Controller Strategy (ks. Etukontrolleri)
- Command Handler Strategy (ks. Sovelluskontrolleri)
- View Handler Strategy (ks. Sovelluskontrolleri)
- Template-based View Strategy (ks. Näkymäavustaja)
- JavaBean Helper Strategy (ks. Näkymäavustaja)

Soveltaminen

Tutkielmaan liittyvässä Kurssinhallintasovelluksessa (liite 1) Palvelu työstäjälle –suunnittelumallia on käytetty siten, että Etukontrolleri-suunnittelumallia edustava EtuKontrolleri-luokka (liite 6) ottaa vastaan palvelupyynnöt ja delegoi kontrollin KurssiSovellusKontrollerille (liite 5), joka edustaa Sovelluskontrolleri-suunnittelumallia. KurssiSovellusKontrolleri-luokka suorittaa avustajan virkaa toimittavan Komento-luokan avulla palvelupyynnön täyttävän operaationnon ja palauttaa sopivalle näkymän osoitteen SovellusKontrollerille. Sovelluskontrolleri

ohjaa suorituksen sopivalle näkymälle. Näkymäavustaja-suunnittelumallin mukaisten avustajien (liite 3) avulla prosessoidaan muotoilulogiikka käyttäen kustomoituja tageja ja tagitiedostoja.

3.2 Liiketoimintakerroksen suunnittelumalleja

Tässä kohdassa esitellään J2EE-arkkitehtuurin liiketoimintakerroksen suunnittelumalleja. Mukana ovat kuvan 1.1 kaikki liiketoimintakerroksen suunnittelumallit. Kustakin suunnittelumallista on kerrottu kuvaus, käyttötarkoitus, vaikutukset ja toteutusstrategioita. Tutkielman rajauksen mukaisesti tutkielmaan liittyvään Kurssinhallintasovellukseen ei ole toteutettu liiketoimintakerroksen suunnittelumalleja, mutta toteutusstrategioiden esittelyn yhteydessä kerrotaan kuinka niitä voisi toteuttaa.

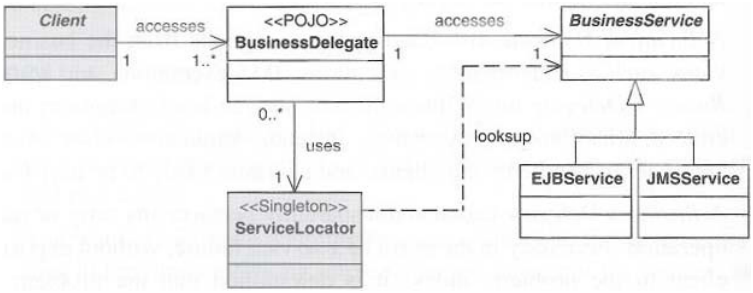
3.2.1 Liiketoimintadelegaatti (Business Delegate)

Liiketoimintadelegaatin avulla Alurin & al. (2003) mukaan kapseloidaan pääsy liiketoimintapalveluihin, kuten Palvelunpaikallistajaan, ja piilotetaan toteutuksen yksityiskohdat. Liiketoimintadelegaatti käyttää Palvelunpaikallistajaa liiketoimintapalvelun etsimiseen. Liiketoimintadelegaatti käsittelee liiketoimintakerroksen poikkeukset ja generoi sovellustason poikkeuksia, joita asiakassovelluksen on helpompi käsitellä. Liiketoimintadelegaatti suorittaa kutsujen uudelleenyritykset ja toipumisoperaatiot palvelun toimintahäiriön aikana. Asiakassovellus saa tiedon toimintahäiriöstä ainoastaan silloin, kun Liiketoimintadelegaatti määrittelee sen ratkaisemattomaksi. Liiketoimintadelegaatti voi kopioida välimuistiin tulokset ja viittaukset liiketoimintapalveluihin, mikä parantaa suorituskykyä vähentämällä toistuvia kutsuja. Neil ja Gill (2003) käyttävät refaktorointina Liiketoimintadelegaattia vähentämään kytkentää esityskerroksen ja liiketoimintakerroksen välillä. Hammouda ja Koskimies (2002) käyttävät Liiketoimintadelegaattia kehittämässään työkalussa, jolla luodaan sovelluksen arkkitehtoninen kehys suunnittelumallien avulla.

Alurin & al. (2003) mukaan Liiketoimintadelegaattia käytetään, kun

- esityskerroksen sovellukset käyttävät liiketoimintakerroksen palveluita
- halutaan vähentää kytkentää asiakassovelluksen ja järjestelmän liiketoimintapalveluiden kanssa ja halutaan piilottaa toteutuksen yksityiskohdat, kuten palveluiden haku.
- halutaan välttää toistuvien etäpalveluiden kutsuminen
- halutaan muuttaa tietoliikenneverkon poikkeuskäsittely sovelluksen tai käyttäjän ymmärtämään muotoon

- halutaan piilottaa palvelun luominen, uudelleenkonfigurointi ja kutsujen uudelleenyritykset asiakassovellukselta.



Kuva 3.17: Liiketoimintadelegaatti (Alur & al., 2003)

BusinessDelegate (kuva 3.17) kontrolloi ja suojaa liiketoimintapalvelua. BusinessDelegate voi saada parametrina kutsuttavan palvelun (EJBHome, EJBService) avaimen asiakassovellukselta. Avaimen avulla BusinessDelegate ottaa uudelleen yhteyttä BusinessServiceen käyttämättä ServiceLocatoria. ServiceLocatoria käytetään palvelun etsimiseen silloin, kun avainta ei ole saatu parametrina.

ServiceLocator on Palvelunpaikallistaja, joka piilottaa palvelun etsimiseen liittyvät yksityiskohdat. BusinessService on liiketoimintakerroksen komponentti, kuten Java-papu, joka tyypillisesti toteutetaan Sessiojulkisivuna tai JMS-komponenttina.

Taulukossa 3.9 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.9: Liiketoimintadelegaatin vaikutuksia.

Vaikutus	Selitys
Vähentää kytkentää ja parantaa ylläpidettävyyttä	Suunnittelumalli vähentää kytkentää esityskerroksen ja liiketoimintakerroksen välillä piilottamalla liiketoimintakerroksen toteutuksen yksityiskohdat (vrt. Välittäjä-suunnittelumalli). Muutosten hallinta helpottuu, koska liiketoimintaa koskevat muutokset voidaan tehdä esityskerrokseen nähden keskitetysti Liiketoimintadelegaatissa.
Parantaa poikkeuskäsittelyä	Liiketoimintadelegaatti muuttaa liiketoimintakerroksesta tulevat poikkeukset, esim. tietoliikenneverkon poikkeukset, asiakassovelluksen ymmärtämään muotoon
Parantaa saatavuutta	Liiketoimintadelegaatti suorittaa uudelleenyritykset ja toipumisoperaatiot, palvelun toimintahäiriön aikana.
Paljastaa yksinkertaisen ja yhdenmukaisen rajapinnan asiakassovellukselle	Liiketoimintadelegaatti toteutetaan yksinkertaisena Java-oliona, jota asiakassovellusten on helpompi käyttää kuin liiketoimintapalvelun kompleksista rakennetta.

Taulukko 3.10: Liiketoimintadelegaatin vaikutuksia (jatk.).

Vaikutus	Selitys
Parantaa suoritustyky	Liiketoimintadelegaatti voi kopioida välimuistiin tietoa esityskerroksen komponentteja varten tehostaakseen yleisimpien palveluiden pyyntöjen käsittelyä.
Tuottaa ylimääräisen tason kerrokseen	Lisättävä taso saattaa näyttää tuovan kompleksisuutta ja alentavan joustavuutta, mutta sen tuomat hyödyt korvaavat haittapuolet.
Piilottaa etäyhteyden	Liiketoimintadelegaatti on asiakassovelluspuolen Edustaja etäpalveluille. Liiketoimintadelegaatti kutsuu liiketoimintapalvelua tietoliikenneverkon kautta.

Liiketoimintadelegaatin toteutuksesta esitellään Alurin & al. (2003) kuvaama strategia, Delegate Proxy Strategy. Strategia on valittu esitettäväksi tutkielmassa, koska se mahdollistaa EJB-teknologian käyttämisen suunnittelumallin kanssa.

Delegate Proxy Strategy

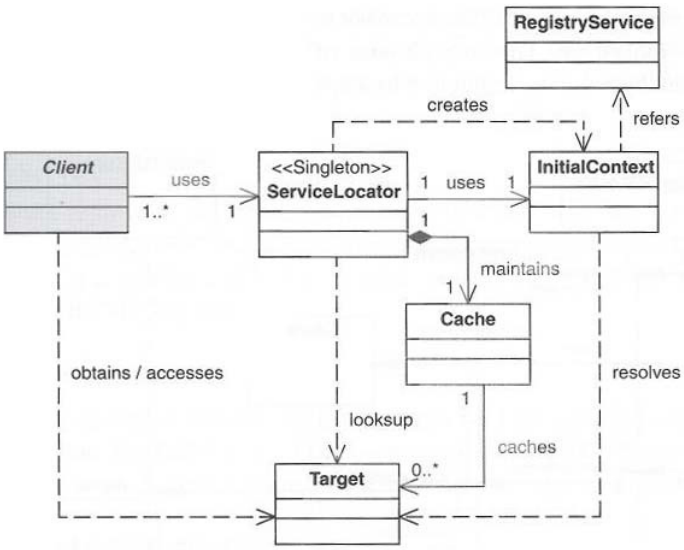
Liiketoimintadelegaatti paljastaa rajapinnan, jonka avulla asiakassovellus pääsee liiketoimintapalvelu-API:in. Liiketoimintadelegaatti toimii liiketoimintapalveluiden Edustajana, ohjaten asiakassovelluksen metodikutsut palveluoliolle. Liiketoimintadelegaattiin voidaan lisätä validointi, liiketoimintatiedon kopiointi välimuistiin ja viittaukset liiketoimintapalveluihin. Tutkielmaan liittyvässä Kurssinhallintasovelluksessa Liiketoimintadelegaatin avulla Komento-oliot voisivat ottaa yhteyden Liiketoimintakerroksen komponentteihin esimerkiksi Kurssin tietojen hakemiseksi ja ylläpitämiseksi.

3.2.2 Palvelunpaikallistaja (Service Locator)

Palvelunpaikallistajan avulla Alurin & al. (2003) mukaan kapseloidaan ja toteutetaan palveluiden ja komponenttien etsintä. Palvelunpaikallistaja piilottaa etsinnän yksityiskohdat ja kapseloi siihen liittyvät riippuvuudet. Palvelunpaikallistaja vähentää koodin kompleksisuutta tarjoamalla keskitetyn paikan, jossa etsintää kontrolloidaan ja parantaa tehokkuutta välimuistin avulla. Tavallisesti järjestelmässä tarvitaan vain yksi Palvelunpaikalllistaja. Voidaan kuitenkin luoda omat paikallistajat esityskerrokseen ja liiketoimintakerrokseen. Neil ja Gill (2003) käyttävät refaktorointina Palvelunpaikallistajaa yksinkertaistamaan liiketoimintapalvelujen kutsumekanismia. Hammouda ja Koskimies (2002) käyttävät Palvelunpaikallistajaa kehittämässään työkalussa, jolla luodaan sovelluksen arkkitehtoninen kehys suunnittelumallien avulla.

Alurin & al. (2003) mukaan Palvelunpaikallistajaa käytetään, kun

- käytetään JNDI API:a liiketoimintakomponenttien, kuten EJB ja JMS, etsimiseen ja käyttämiseen
- halutaan keskittää ja uudelleenkäyttää etsintämekanismin toteutusta
- halutaan kapseloida valmistajakohtaiset riippuvuudet komponenttien rekisteröintiin ja piilottaa riippuvuudet ja kompleksisuus asiakassovelluksilta
- halutaan välttää alustamisen ja komponenttien etsinnän aiheuttamaa kuormitusta
- halutaan uudelleenmuodostaa yhteys aikaisemmin käytettyyn pavun intanssiin käyttäen sen kahvaa (handle).



Kuva 3.18: Palvelunpaikallistaja (Alur & al., 2003)

Client (kuva 3.18) on Palvelunpaikallistajan asiakassovellus, jonka tarvitsee etsiä ja käyttää liiketoiminta- tai integrointikerroksen komponenttia tai palvelua. Esim. Liiketoimintadelegaatti on Palvelunpaikallistajan asiakas etsiessään Sessiojulkisivua. Samoin Tiedonhakuolio on paikallistajan asiakas käyttäessään Palvelunpaikallistajaa löytääkseen JDBC DataSource –instanssin.

ServiceLocator kapseloi API lookup (naming) -palvelut, valmistajakohtaiset riippuvuudet, etsinnän kompleksisuuden, Liiketoimintaolion luonnin ja tarjoaa yksinkertaisen rajapinnan asiakassovellukselle.

Cache on valinnainen ServiceLocator, joka pitää muistissa aikaisemmin suoritetut paikallistamiset. Cache optimoi ServiceLocatorin käyttöä vähentämällä usein toistuvien paikallistamisien uudelleen suorittamista.

InitialContext on paikallistamis- ja luomisprosessin lähtötilanne. Palveluntarjojat tarjoavat kontekstiolion, joka vaihtelee riippuen siitä minkä tyyppistä kohdetta (Target) kulloinkin etsitään ServiceLocatorilla. ServiceLocator, joka tarjoaa palveluita erityylisille Target-komponenteille (EJB, JMS, jne.), käyttää vastaavan tyyppistä kontekstioliota, joka saadaan palveluntarjoajalta. Esim. EJB-sovelluspalvelimen kontekstin tarjoaja voi olla eri kuin JMS- palvelun kontekstin tarjoaja. Target on liiketoiminta- tai integrointikerroksen palvelu tai komponentti, jota Client paikallistaa ServiceLocatorin avulla. Esim. kun Client etsii EJB-komponenttia, Target on EJB Home -olio. Target voi olla myös JDBC DataSource tai JMS ConnectionFactory, kuten TopicConnectionFactory tai QueueConnectionFactory.

RegistryService on rekisteröinnin toteutus, joka säilyttää viittaukset palveluihin tai komponentteihin, jotka on rekisteröity asiakassovelluksen palvelun tuottajiksi. RegistryService on julkaisu- ja etsintäpalvelu, kuten JNDI- tai UDDI-arkisto tai eBXML RegRep.

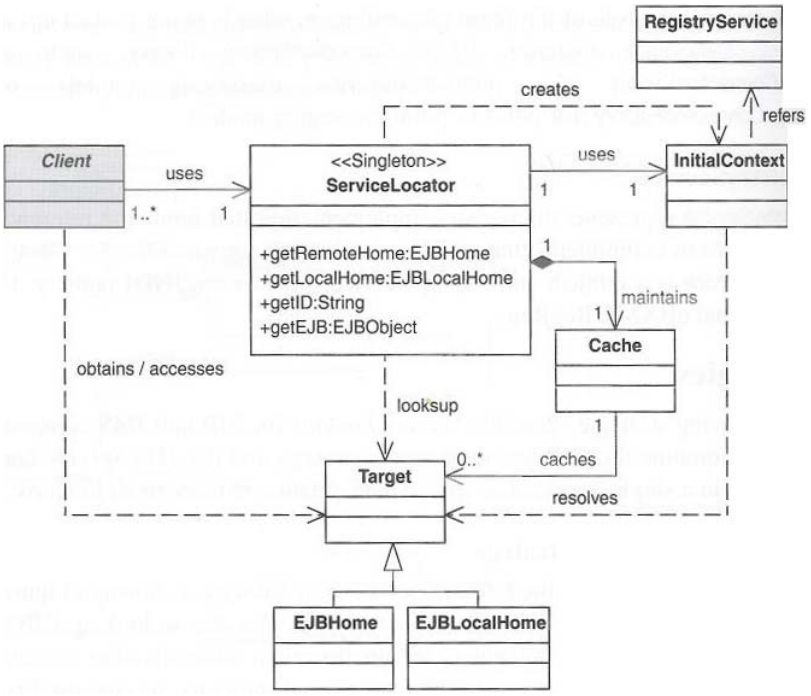
Taulukossa 3.10 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.11: Palvelunpaikallistajan vaikutuksia.

Vaikutus	Selitys
Abstrahoi kompleksisuuden	Palvelunpaikallistaja piilottaa ja kapseloi palvelun etsinnän kompleksisuuden ja luontiprosessin asiakassovellukselta.
Yhdenmukainen pääsy palveluihin	Palvelunpaikallistaja tarjoaa yhdenmukaisen rajapinnan asiakassovelluksille, mikä vähentää kehitys- ja ylläpitotyön kuormaa.
Helpottaa EJB-Liiketoiminta-olioiden lisäämistä	Koska asiakassovellukset eivät ole tietoisia EJB Home-olioista, niitä voidaan lisätä jälkeenpäin vaikuttamatta asiakassovelluksiin mitenkään. JMS-asiakassovellukset eivät ole tietoisia JMS-yhteystehtaista, joten uusia tehtaita voi lisätä vaikuttamatta mitenkään asiakassovelluksiin.
Vähentää tietoliikenneverkon kuormittumista	Asiakassovellukset eivät sekaannu etsintä- ja luontiprosesseihin. Koska Palvelunpaikallistaja tekee tämän työn, se voi yhdistää Liiketoimintaolioiden etsintään tarvittavia tietoliikenneverkon kutsuja.
Kehittää asiakassovelluksen suorituskykyä välimuistin avulla	Palvelunpaikallistaja voi kopioida välimuistiin alkuperäiset kontekstioliot ja viittaukset tehdasolioihin (EJB Home, JMS connection factory). Palvelunpaikallistaja voi varastoida myös WSDL-määritykset ja päätepisteet, kun otetaan yhteys verkkopalveluihin (web services).

Palvelunpaikallistajan toteutuksesta esitellään Alurin & al. (2003) kuvaama strategia, EJB Service Locator Strategy. Strategia on valittu esitettäväksi tutkielmassa, koska se mahdollistaa EJB-teknologian käyttämisen suunnittelumallin kanssa.

EJB Service Locator Strategy



Kuva 3.19: EJB Service Locator Strategy (Alur & al., 2003)

Strategiassa J2EE-ympäristössä EJB-asiakassovellukset voivat käyttää Palvelunpaikallistajaa EJB-komponenttien etsintään. JNDI-ympäristö on asennettu niin, että se ottaa yhteyttä sovelluksen nimi- ja hakemistopalveluun. Client (kuva 3.19) käyttää InitialContextia sovelluksen JNDI-arkistopalvelun paikallistamiseen (RegistryService) ja etsii palvelupyynnön mukaista EJB Home -oliota (Target) käyttäen rekisteröityä JNDI-nimeä. Kun EJB Home -olio on saatu, se varastoidaan Palvelunpaikallistajaan, josta se saadaan tarvittaessa ilman uutta JNDI-hakua. Asiakassovellus voi EJB Home -olion avulla luoda, poistaa tai etsiä EJB-instansseja. EJB-Palvelunpaikallistaja voi etsiä sekä paikallisia että etäpapuja. Tutkielmaan liittyvässä Kurssinhallintasovelluksessa Palvelunpaikallistajan avulla voisi etsiä tarkoitukseen sopivan palvelun, esimerkiksi KurssiPapu-entiteettipavun, kurssin tietojen hakemiseksi ja ylläpitämiseksi.

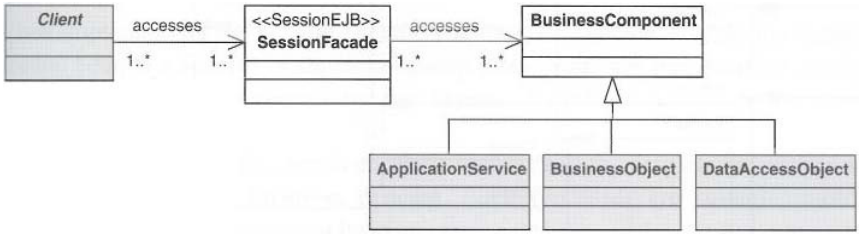
3.2.3 Sessiojulkisivu (Session Façade)

Sessiojulkisivun avulla Alurin & al. (2003) mukaan kapseloidaan liiketoimintakerroksen komponentit ja paljastetaan karkearakenteinen palvelurajapinta asiakassovelluksille. Sessiojulkisivun kautta otetaan yhteys liiketoimintakomponentteihin. Sessiojulkisivu toteutetaan sessiopapuna, joka kommunikoi esim. Liiketoimintaolioiden ja Sovelluspalveluiden kanssa. Liiketoimintalogiikka tulee sijoittaa Sessiojulkisivun sijasta Sovelluspalveluun, jota Sessiojulkisivu kutsuu. Neil ja Gill (2003) käyttävät refaktorointina Sessiojulkisivua yhdenmukaistamaan pääsyä liiketoimintakerrokseen, vähentämään kytkentää liiketoimintaolioiden ja asiakassovelluksen välillä ja vähentämään kutsujen määrää asiakkaan ja palvelimen välillä. Hammouda ja Koskimies (2002) käyttävät Sessiojulkisivua kehittämässään työkalussa, jolla luodaan sovelluksen arkkitehtoninen kehys suunnittelumallien avulla. Myös Chen ja MacDonald (2005) ovat kehittäneet vastaavanlaisen työkalun, jossa on käytetty Sessiojulkisivua.

Jokaista käyttötapausta varten ei ole kannattavaa luoda omaa Sessiojulkisivua. Sen sijaan toisiinsa suhteessa olevat käyttötapaukset voidaan koota ryhmäksi, jonka yksi Sessiojulkisivun ilmentymä kapseloi. Esim. KurssiSessiojulkisivu kapseloisi käyttötapaukset ”luo uusi kurssi”, ”muokkaa kurssia”, ”näytä kurssiin tiedot” jne.

Alurin & al. (2003) mukaan Sessiojulkisivua käytetään, kun

- halutaan välttää tiukat kytkennät asiakassovelluksen ja liiketoimintakerroksen komponenttien välillä, ts. asiakassovellusten suorat kontaktit liiketoimintakerroksen komponentteihin
- halutaan etäyhteyskerros Liiketoimintaolioihin ja muihin liiketoimintakerroksen komponentteihin
- halutaan yhdistää ja paljastaa Sovelluspalvelut ja muut palvelut etäasiakkaille
- halutaan keskittää ja yhdistää asiakassovellukselle paljastettava liiketoimintalogiikka
- halutaan piilottaa kompleksinen vuorovaikutus ja sisäiset suhteet liiketoimintakomponenttien ja palveluiden välillä.



Kuva 3.20: Sessiojulkisivu (Alur & al., 2003)

Client (kuva 3.20) on asiakassovellus (tyypillisesti Liiketoimintadelegaatti), joka tarvitsee liiketoimintapalvelua. SessionFacade toteutetaan sessiopapuna, joka tarjoaa karkearakenteisen korkeamman tason palvelun abstraktion asiakassovellukselle. SessionFacade piilottaa kompleksisuuden käsitellä useita käyttötapaukseen liittyviä BusinessComponentteja. BusinessComponent osallistuu asiakassovelluksen tekemän palvelupyynnön täyttämiseen. BusinessComponent voidaan toteuttaa joko Liiketoimintaoliona, joka mallintaa liiketoimintatietoa, tai Sovelluspalveluna.

ApplicationService kapseloi BusinessObjectit ja toteuttaa liiketoimintalogiikan, joka suoritetaan pyydetyn palvelun saamiseksi. SessionFacade kommunikoi useiden ApplicationService-olioiden kanssa täyttääkseen asiakassovellukselta tulleen palvelupyynnön. DataAccessObject esittää tässä tapausta, jolloin Sessiojulkisivu voi suoraan käyttää Tiedonhakuoliota saadakseen liiketoimintatietoa.

Taulukossa 3.11 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.12: Sessiojulkisivun vaikutuksia.

Vaikutus	Selitys
Paljastaa yhdenmukaisen karkearakenteisen rajapinnan	Sessiojulkisivu kapseloi monimutkaisen liiketoimintakomponenttien vuorovaikutuksen ja tarjoaa yksinkertaisen karkearakenteisen palvelukerroksen rajapinnan.
Vähentää kytkentää kerrosten välillä	Sessiojulkisivu poistaa tiukat kytkökset ja riippuvuudet esitys- ja liiketoimintakerrosten väliltä. (vrt. Välittäjä-suunnittelumalli)
Edistää kerrostamista ja lisää joustavuutta sekä ylläpidettävyyttä	Käyttämällä Sessiojulkisivuja Sovellupalveluiden kanssa, järjestelmän joustavuus lisääntyy kerrostamalla ja keskittämällä palveluita. Tämä parantaa järjestelmän kykyä mukautua tarvittaviin muutoksiin vähentyneen kytkennän myötä. Joskin muutokset liiketoimintalogiikkaan saattavat aiheuttaa muutoksia Sovelluspalveluihin tai jopa Sessiojulkisivuihin, kerrostaminen tekee muutokset helpommin hallittavaksi.
Vähentää kompleksisuutta	Käyttämällä Sovelluspalveluita, vähennetään Sessiojulkisivun kompleksisuutta. Liiketoimintadelegaatin käyttäminen vähentää asiakassovelluksen koodin kompleksisuutta. Järjestelmän ylläpidettävyyden ja joustavuuden kehittäminen helpottuu.

Taulukko 3.13: Sessiojulkisivun vaikutuksia (jatk.).

Vaikutus	Selitys
Parantaa suorituskykyä ja vähentää hienorakenteisia etämetodeja	Sessiojulkisivu parantaa suorituskykyä vähentyneiden etäkutsujen myötä, yhdistämällä useita hienorakenteisia vuorovaikutuksia yhteen karkearakenteiseen metodiin. Tavallisesti Sessiojulkisivut sijaitsevat samassa prosessissa liiketoimintakomponenttien kanssa mahdollistaen nopeamman kommunikoinnin niiden välillä.
Keskittää turvallisuuden hallinnan	On helpompaa ja hallittavampaa määrittää turvallisuuteen liittyvät menetelmät Sessiojulkisivun yhteydessä, kuin jokaisessa Liiketoimintaoliossa, koska asiakassovellukset ottavat yhteyttä Sessiojulkisivuun.
Tiedonvaihtamisen kontrolloinnin keskittäminen	Sessiojulkisivu tarjoaa karkearakenteisen etäyhteyden liiketoimintakerroksen palveluihin, joten tiedonvaihtamisen hallinnan keskittäminen on helpompaa Sessiojulkisivukerrokseen.
Paljastaa vähemmän etärajapintoja asiakassovelluksille	Sessiojulkisivu tarjoaa karkearakenteisen yhteyden liiketoimintakomponentteihin, mikä vähentää asiakassovellukselle paljastettujen liiketoimintakomponettien määrää. Vuorovaikutuksien määrä asiakassovelluksien ja Sessiojulkisivun välillä on pienempi kuin suorien yhteyksien määrä asiakassovelluksien ja yksittäisten liiketoimintakomponenttien välillä.

Sessiojulkisivun toteutuksesta esitellään kaksi Alurin & al. (2003) kuvaamaa strategiaa, Stateless Session Facade Strategy ja Stateful Session Facade Strategy. Strategiat on valittu esitettäväksi tutkielmassa, koska ne mahdollistavat EJB-teknologian käyttämisen suunnittelumallin kanssa.

Stateless Session Facade Strategy

Liiketoimintaprosessi, joka tarvitsee vain yhden metodikutsun palvelupyynnön täyttämiseksi, on ei-keskusteleva liiketoimintaprosessi. Metodikutsujen välillä ei tarvitse tallettaa keskustelun tilaa, joten prosessi on tarkoituksenmukaisempaa toteuttaa tilattomana sessiopapuna. Tutkielmaan liittyvässä Kurssinhallintasovelluksessa kurssitietojen hakemisen voisi toteuttaa tilattomana sessiopapuna.

Stateful Session Facade Strategy

Liiketoimintaprosessi, joka tarvitsee useita metodikutsuja palvelupyynnön täyttämiseksi, on keskusteleva liiketoimintaprosessi. Keskustelun tila täytyy tallettaa asiakkaan metodikutsujen välillä, joten sessiopapu on tarkoituksenmukaisempaa toteuttaa tilallisena sessiopapuna. Tutkielmaan liittyvässä Kurssinhallintasovelluksessa Sessiojulkisivua voisi käyttää silloin, jos esimerkiksi käyttäjän tulisi syöttää kurssin tietoja useammassa vaiheessa ja annetuille syötteille tulisi tehdä tarkistuksia muihin kurssitietoihin tai aika- ja tilavarauksia silmällä pitäen.

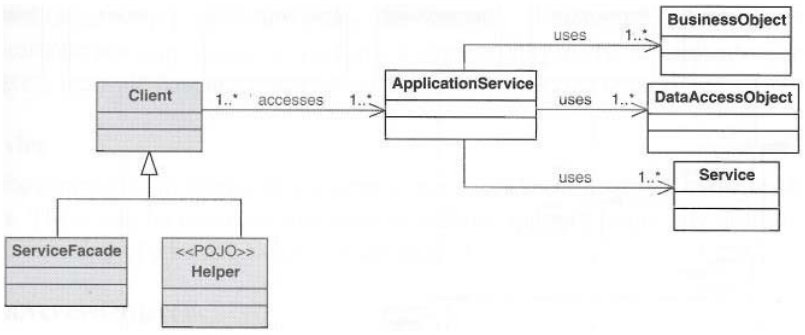
3.2.4 Sovelluspalvelu (Application Service)

Sovelluspalvelu tarjoaa yhdenmukaisen palvelukerroksen keskittämällä ja yhdistämällä Liiketoimintaolioita ja palveluita. Sovelluspalvelun avulla kapseloidaan korkeamman tason liiketoimintalogiikka erilliseen komponenttiin. Sovelluspalvelu tarjoaa infrastruktuurin Sessiojulkisivuille, kuten palveluiden julkisivuille (ServiceFacade, ks. kuva 3.21). Palvelun julkisivu ei tavallisesti sisällä liiketoimintalogiikkaa. Sovelluspalvelun avulla palveluiden julkisivuista tulee yksinkertaisempia, koska ne voivat delegoida liiketoimintakäsittelyn Sovelluspalveluille. Sovelluspalvelu kapseloi liiketoimintalogiikan, joka koskee useita liiketoimintaolioita. Liiketoimintaolio kapseloi liiketoimintalogiikan, joka koskee sen itsensä sisältämää liiketoimintatietoa. Sovelluspalvelua voidaan käyttää myös ulkoisen palvelun, kuten sähköpostipalvelun, käyttämiseen.

Alurin & al. (2003) mukaan Sovelluspalvelu-suunnittelumallia käytetään, kun

- halutaan minimoida Sessiojulkisivun liiketoimintalogiikka
- liiketoimintalogiikka käyttää useita Liiketoimintaolioita tai palveluita
- halutaan tarjota karkearakenteinen palvelu-API liiketoimintakerroksen komponentteihin ja palveluihin
- halutaan kapseloida käyttötapauskohtainen logiikka Liiketoimintaolioden ulkopuolelle.

Asiakassovellus kutsuu Sovelluspalvelun liiketoimintametodia. Liiketoimintametodi voi käyttää useita Liiketoimintaolioita, Tiedonhakuolioita ja muita Sovelluspalveluita, sekä suorittaa tarvittavan liiketoimintalogiikan tuottaakseen kutsutun palvelun.



Kuva 3.21: Sovelluspalvelu (Alur & al., 2003)

Client (kuva 3.21) on asiakassovellus, esim. palvelun julkisivu (Sessiojulkisivu), Sovelluspalvelu tai avustava olio. ApplicationService kapseloi liiketoimintalogiikan, jolla palvelu tuotetaan. Liiketoimintalogiikka voi olla yleistä, useampaa BusinessObjectia koskevaa, käyttötapausta koskevaa tai tietyn tyyppiselle asiakassovellukselle tai kanavalle, kuten web-sovellukselle tai web-palvelulle, tarkoitettua logiikkaa.

BusinessObject on Liiketoimintaolio, jota ApplicationService kutsuu suorittaakseen palvelun. Tavallisesti ApplicationService kutsuu useita BusinessObjecteja yhden palvelun suorittamiseksi.

Service on mielivaltainen komponentti, joka tuottaa palveluita asiakassovelluksille. Komponentti voi olla yleinen funktio, joka avustaa liiketoimintakerroksen palvelupyynnön prosessoimisessa.

DataAccessObject on Tiedonhakuolio niitä tapauksia varten, joissa ApplicationService pääsee suoraan tietolähteen liiketoimintatietoon käyttämättä Liiketoimintaoliota.

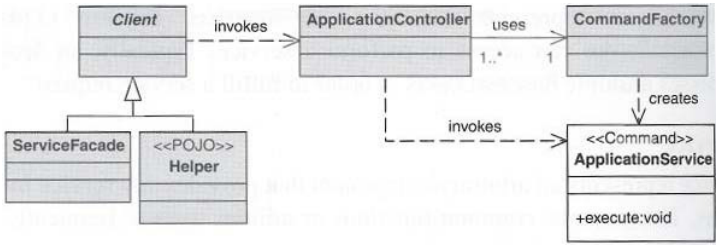
Taulukossa 3.21 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.14: Sovelluspalvelun vaikutuksia.

Vaikutus	Selitys
Keskittää uudelleenkäytettävän liiketoimintalogiikan	Sovelluspalvelut luovat palveluiden kerroksen. Keskitetty kerros kapseloi yleisen liiketoimintalogiikan, joka koskee useita Liiketoimintaolioita.
Parantaa liiketoimintalogiikan uudelleenkäytettävyyttä	Sovelluspalvelut luovat joukon komponentteja, joita voidaan käyttää useissa erilaisissa käyttötapauksissa. Sovelluspalvelut kapseloivat sisäiset Liiketoimintaolion operaatiot.
Välttää koodin kopioitumisen	Luomalla keskitetyn ja uudelleenkäyttettävän liiketoimintalogiikkakerroksen, Sovelluspalvelujen avulla vältetään koodin kopioituminen asiakassovelluksissa, kuten julkisivuissa, avustajissa ja muissa Sovelluspalveluissa.
Yksinkertaistaa julkisivun toteutusta	Liiketoimintalogiikka siirretään pois palvelun julkisivuista. Julkisivuista tulee yksinkertaisempia, koska ne ovat vastuussa vain Sovelluspalveluiden vuorovaikutuksen yhdistämisestä ja delegoimisesta.
Esittelee ylimääräisen tason liiketoimintakerroksessa	Sovelluspalvelut luo ylimääräisen tason liiketoimintakerrokseen, mutta se tarjoaa yleisen liiketoimintalogiikan uudelleenkäytettävyyden.

Sovelluspalvelun toteutuksesta esitellään kaksi Alurin & al. (2003) kuvaamaa strategiaa, Application Service Command Strategy ja GoF Strategy for Application Service Strategy. Strategiat on valittu esitettäväksi tutkielmassa, koska niiden toiminnallinen mekanismi on samankaltainen kuin Sovelluskontrollerin Command Handler –strategiassa. Eli koodia voidaan uudelleenkäyttää.

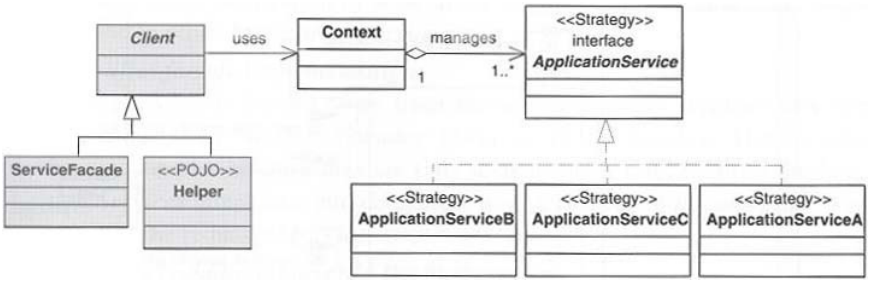
Application Service Command Strategy



Kuva 3.22: Application Service Command Strategy (Alur & al., 2003)

Strategiassa Sovelluspalvelu voidaan toteuttaa käyttäen Komento-suunnittelumallia. Sovelluskontrolleri (ApplicationController) voi tuottaa palvelupyynnön käsittelymekanismin. Client (kuva 3.22) kutsuu ApplicationControlleria, joka käyttää CommandFactorya saadakseen sopivan ApplicationServicen, jonka execute()-metodia kutsumalla suoritetaan palvelupyyntöä vastaava operaatio. Tutkielmaan liittyvässä Kurssinhallintasovelluksessa strategiaa voisi soveltaa kurssin tietojen päivittämisen yhteydessä, mikäli tietoja tulee verrata muihin kurssitietoihin esimerkiksi päällekkäisyyksien tarkistamiseksi. KurssiSessiojulkisivu kutsuisi yhtä tai useampaa KurssiSovelluspalvelua, joka tarkistaa onko esimerkiksi kurssille varattu tila vapaana.

GoF Strategy for Application Service Strategy



Kuva 3.23: GoF Strategy for Application Service Strategy (Alur & al., 2003)

Sovelluspalvelustrategiassa voidaan Sovelluspalvelu toteuttaa käyttäen yleistä Strategia-suunnittelumallia. Sovelluspavelustrategia on ideaali, kun samasta palvelusta on olemassa useita variaatioita. Client (kuva 3.23) käyttää Contextia, joka hallinnoi ApplicationService-rajapinnan toteuttavia strategioita. ApplicationService-rajapinnan avulla suoritetaan tarkoitukseen sopiva algoritmi.

3.2.5 Liiketoimintaolio (Business Object)

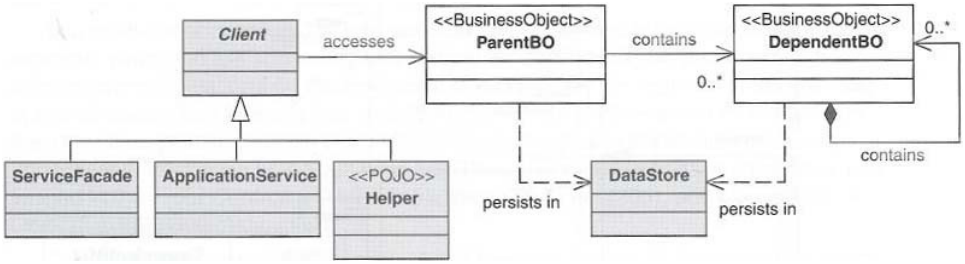
Liiketoimintaolion avulla Alurin & al. (2003) mukaan erotetaan liiketoimintatieto ja -logiikka käyttäen oliomallia. Liiketoimintaolio kapseloi ja käsittelee liiketoimintatietoa, -logiikkaa ja tiedon säilyttämistä. Logiikka, joka koskee useita Liiketoimintaolioita, toteutetaan käyttäen Sovelluspalvelua ja Sessiojulkisivua. Jacyntho & al. (2002) sijoittavat Liiketoimintaolion mallikerrokseen MVC-mallissa.

Liiketoimintaolio toteuttaa joko tavallisena Java-oliona tai entiteettipapuna. Kummassakin tapauksessa on otettava huomioon vaatimukset turvallisuudelle, tiedonvaihdon hallinnalle, allas-mekanismille, välimuistiin kopioinnille ja yhtäaikaiselle käytölle. Suositeltavaa on että asiakassovellukset käyttävät Sessiojulkisivua ja Sovelluspalvelua kommunikointiin Liiketoimintaolioiden kanssa. Suorat kutsut Liiketoimintaolioihin johtavat hienorakenteisiin vuorovaikutuksiin asiakassovelluksen ja Liiketoimintaolioiden välillä. Toteutettaessa Liiketoimintaolio entiteettipapuna, käytetään hyväksi Komposiittientiteettiä ja säilytykseen joko BMP (bean-managed persistence)- tai CMP (container-managed persistence) -mekanismia.

Alurin & al. (2003) mukaan Liiketoimintaoliota käytetään, kun

- järjestelmän käsitteellinen malli sisältää toisiinsa suhteessa olevia komposiittiolioita
- järjestelmän käsitteellisessä mallissa on monimutkainen liiketoimintalogiikka
- halutaan erottaa liiketoimintatila ja –logiikka muusta sovelluksesta
- halutaan keskittää sovelluksen liiketoimintalogiikka ja –tila
- halutaan kehittää liiketoimintalogiikan uudelleenkäytettävyyttä ja välttää koodin kopioitumista.

Client (kuva 3.20) kutsuu ParentBO:n liiketoimintametodia, joka lataa tietonsa tietovarastosta. ParentBO suorittaa itseään koskevan liiketoimintalogiikan kohdistaen sen ladattuun tietoon. ParentBO tallentaa tietonsa tietovarastoon, jos muutoksia säilytettävään tietoon tarvitsee tehdä. ParentBO voi kutsua siitä riippuvaisia liiketoimintaolioita (DependentBO) täyttääkseen



Kuva 3.24: Liiketoimintaolio (Alur & al., 2003)

asiakassovelluksen tekemän palvelupyynnön. Jokainen näistä riippuvaisista olioista suorittaa oman liiketoimintalogiikkansa ja –sääntönsä ja kommunikoi tarvittaessa tietovaraston kanssa.

Taulukossa 3.13 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.15: Liiketoimintaolion vaikutuksia.

Vaikutus	Selitys
Oliosuuntautunut lähestymistapa	Liiketoimintaoliot luovat loogisen kerroksen, joka heijastaa oliomallin mukaista toteutusta liiketoimintamallista.
Keskittää liiketoimintakäyttäytymistä ja –tilaa, sekä kehittää uudelleenkäyttöä	Liiketoimintaoliot tuottavat keskitetyn ja modulaarisen lähestymistavan monikerrosarkkitehtuuriin abstrahoimalla ja toteuttamalla liiketoimintalogiikan, -säännöt ja –käyttäytymisen erillisenä komponenttijoukkona. Tämä mahdollistaa abstraktioiden uudelleenkäytön liiketoimintakerroksessa erilaisissa käyttötapauksissa ja erilaisten asiakassovellusten kanssa.
Kehittää koodin ylläpidettävyyttä ja vähentää sen kopioitumista	Liiketoimintatilan ja -käyttäymisen keskittämisen myötä, asiakassovellukseen ei tarvitse upottaa liiketoimintalogiikkaa. Siten vältetään koodin toistuminen muualla sovelluksessa. Liiketoimintaolioiden käyttäminen kehittää järjestelmän ylläpidettävyyttä kokonaisuudessaan, koska Liiketoimintaoliot edistävät uudelleenkäytettävyyttä ja koodin keskittämistä.
Erottaa säilyttämislogiikan liiketoimintalogiikasta	Säilytysmismekanismi voidaan piilottaa ja erottaa Liiketoimintaolioista. Säilytysstrategiana voidaan käyttää esim. entiteettipapuja.
Edistää palvelusuuntautunutta arkkitehtuuria	Liiketoimintaoliot käyttäytyvät keskitettynä oliomallina asiakkaille sovelluksessa. Liiketoimintaoliolle voi rakentaa erilaisia palveluita, jotka voivat käyttää toisia palveluita esim. säilytys, liiketoimintasäännöt, integrointi jne.
Lisää ylimääräisen tason kerrokseen	Joissakin yksinkertaisissa sovelluksissa, tiukka erotteleminen esitys-, liiketoiminta- ja integrointikerroksen välillä voi olla enemmänkin muodollisuutta kuin välttämättömyyttä. Etenkin sovelluksissa, joissa on yksinkertainen liiketoimintamalli ja –logiikka, voi olla helpompaa että esityskerroksen komponentit pääsevät kutsumaan suoraan integrointikerroksen tiedonhakuolioita. Sovelluskehityksen myöhemmässä vaiheessa voidaan kuitenkin tulla tilanteeseen, jossa aiemmin riittävä tietomalli ei jousta tarvittaviin uudistuksiin. Tilanteen korjaaminen jälkikäteen voi tulla kalliiksi.
Voi johtaa paisuneisiin olioihin	Tietyt käyttötapaukset vaativat Liiketoimintaoliolta ainoastaan siihen kapseloitua olennaista käyttäytymistä. Liiketoimintaoliot ovat taipuvaisia paisumaan, kun käyttötapauksia kohdistetaan siihen enemmän. Paisuminen vältetään toteuttamalla Liiketoimintaolioon luontaisesti kuulumaton logiikka Sovelluspalveluun.

Liiketoimintaolion toteutuksesta esitellään Alurin & al. (2003) kuvaama strategia, Composite Entity Business Object Strategy. Strategia on valittu esitettäväksi tutkielmassa, koska se mahdollistaa EJB-teknologian käyttämisen suunnittelumallin kanssa.

Composite Entity Business Object Strategy

Strategiassa Liiketoimintaoliot voidaan toteuttaa entiteettipapuina käyttäen Komposiittientiteettiä. Entiteettipavut toteutetaan paikallisina papuina ja käyttäen säilytykseen CMP-mekanismia. Liiketoimintaoliot kapseloidaan Sessiojulkisivun avulla. Tutkielmaan liittyvässä Kurssinhallintasovelluksessa Kurssi-, Opettaja- ja Opiskelija-luokat voisi toteuttaa entiteettipapuina, joita käytetään Sessiojulkisivun kautta.

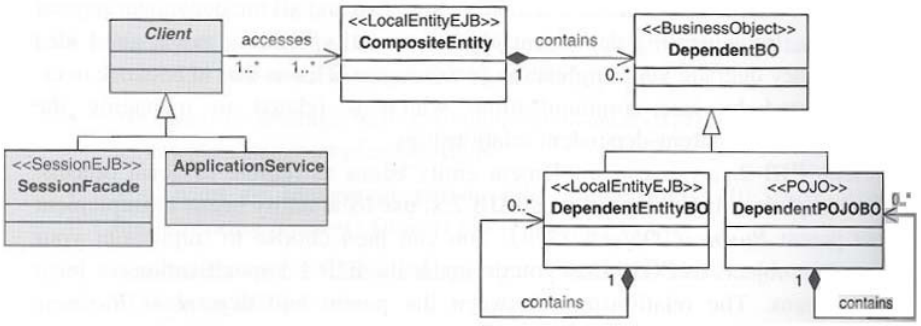
3.2.6 Komposiittientiteetti (Composite Entity)

Komposiittientiteetin avulla Alurin & al. (2003) mukaan pysyvät Liiketoimintaoliot toteutetaan paikallisina entiteettipapuina tai tavallisina Java-olioina. EJB:n myötä voi hyödyntää sen tuomia ominaisuuksia, kuten säiliön hallitsemat tapahtumat, turvallisuus ja säilytys. Komposiittientiteetti yhdistää joukon toisiinsa vanhempi-lapsi-suhteessa olevia Liiketoimintaolioita karkearakenteiseksi entiteettipapujen toteutukseksi. Komposiittientiteettipapu voi toteuttaa vanhemman olion ja kaikki siihen lapsi-suhteessa olevat riippuvaiset oliot (ks. kuva 3.24). Suhteet vanhemman ja siitä riippuvaisen Liiketoimintaolion välillä voidaan toteuttaa CMR (container-managed relationships) -mekanismia käyttäen. Säilytykseen käytetään CMP (container-managed persistence) -mekanismia. Asiakassovellus ei käytä suoraan entiteettipapuja, vaan se ottaa yhteyden Sessiojulkisivuun käyttääkseen entiteettipapuja.

Alurin & al. (2003) mukaan Komposiittientiteettiä käytetään, kun

- halutaan välttää etäentiteettipapujen haittavaikutuksia, kuten tietoliikenneverkon kuormitusta
- halutaan toteuttaa vanhempi-lapsi-suhteet tehokkaasti, kun käytetään Liiketoimintaolioita entiteettipapuina
- halutaan kapseloida ja yhdistää tavallisina Java-olioina toteutetut Liiketoimintaoliot entiteettipapujen avulla
- halutaan hyödyntää EJB-säiliön hallitsemia tapahtuma- ja turvallisuusominaisuuksia
- halutaan kapseloida fyysinen tietokanta asiakassovelluksilta

Client (kuva 3.25) on asiakas, joka voi olla sessiopapu-tyyppinen Sessiojulkisivu (SessionFacade), tai Sovelluspalvelu (ApplicationService). CompositeEntity on karkearakenteinen entiteettipapu, joka sisältää siitä riippuvaisia Liiketoimintaolioita.



Kuva 3.25: Komposiittientiteetti (Alur & al., 2003)

DependentBO, DependentEntityBO ja DependentPOJOBO ovat riippuvaisia vanhemmastaan, joka säätelee niiden elinkaarta. Riippuvainen olio voi sisältää toisia riippuvaisia olioita. Riippuvainen olio voidaan toteuttaa joko paikallisena entiteettipapuina (DependentBO ja DependentEntityBO) tai tavallisena Java-oliona (DependentPOJOBO).

Taulukossa 3.14 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

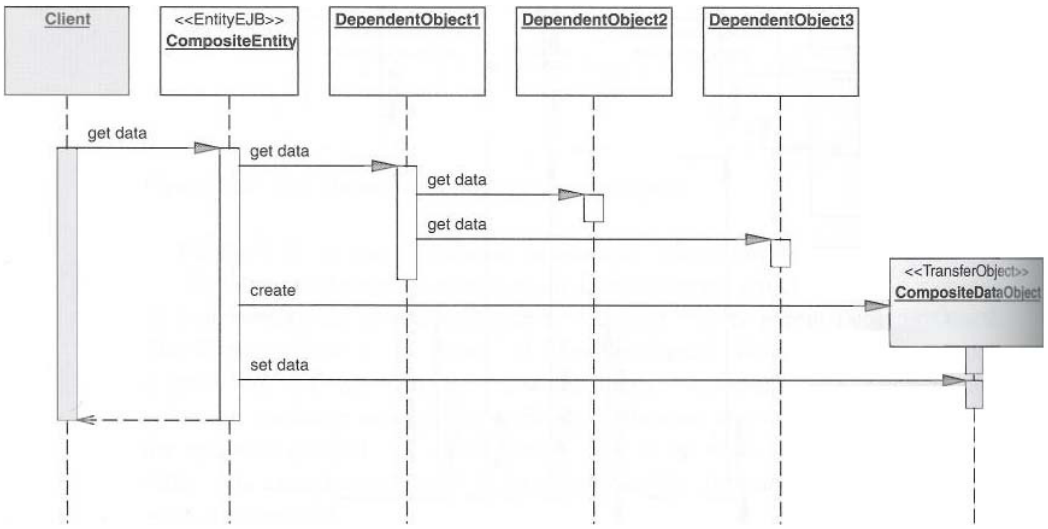
Taulukko 3.16: Komposiittientiteetin vaikutuksia.

Vaikutus	Selitys
Lisää sovelluksen ylläpidettävyyttä	Käytettäessä EJB:n versiota 2.x, riippuvaiset Liiketoimintaoliot voidaan toteuttaa paikallisina entiteettipapuina ja hyödyntää CMR:n ja CMP:n suhde- ja säilytysominaisuuksia.
Parantaa tietoliikenneverkon suorituskykyä	Liiketoimintaolioiden toteuttaminen Komposiittientiteeteiksi vähentää verkon kuormitusta, koska asiakassovelluksen yhteydet entiteettipapuihin ovat paikallisia. Tavallisten Java-olioiden käyttäminen on silti tehokkaampaa kuin paikallisten entiteettipapujen, koska papujen säiliöpalvelut elinkaaren ylläpitämiseen, turvallisuuteen ja tiedonvaihdon hallintaan vievät aina resursseja, vaikka niitä ei kaikissa tapauksissa tarvittaisikaan.
Vähentää riippuvuutta tietokantaskeemaan	Komposiittientiteetti tuottaa oliomaisen näkymän tietokannan sisältämään tietoon. Tietokantaskeema on piilossa asiakassovelluksilta ja se on mapattu sisäisesti komposiittiin entiteettiin. Muutokset tietokantaskeemaan voivat aiheuttaa muutostarpeita komposiittiin entiteettipapuun. Muutokset eivät kuitenkaan koske asiakassovelluksia, sillä komposiitit entiteettipavut eivät paljasta skeemaa ulkopuolelle.
Lisää olioiden rakeisuutta	Asiakassovellus käyttää vanhempaa entiteettipapua, hienorakenteisten riippuvaisten entiteettipapujen sijaan. Vanhempi entiteettipapu piilottaa riippuvaisten olioiden kompleksisuuden ja paljastaa yksinkertaisemman rajapinnan asiakkaalle. Komposiittientiteetin avulla vältetään hienorakenteisiin olioihin kohdistuvat metodikutsut ja siten suunnittelumalli vähentää tietoliikenteen kuormitusta.
Helpottaa komposiitin Kuljetusolion luomista	Komposiittientiteetti voi luoda Kuljetusolion, joka sisältää kaikki entiteettipapujen ja siitä riippuvaisten olioden tiedot, ja palauttaa sen asiakassovellukselle.

Komposiittientiteetin toteutuksesta esitellään Alurin & al. (2003) kuvaama strategia, Composite Transfer Object Strategy. Strategia on valittu esitettäväksi tutkielmassa, koska se mahdollistaa EJB-teknologian käyttämisen suunnittelumallin kanssa.

Composite Transfer Object Strategy

Strategiassa (kuva 3.26) Client (joka on yleensä Sessiojulkisivu) hakee kaiken tarvitsemansa tiedon yhden metodikutsun avulla. Komposiittientiteetti toteuttaa vanhemman Liiketoiminta-olion ja hierarkian riippuvaisia olioita. Komposiittientiteetti muodostaa serialisoitavan Kuljetusolion olioiden sisältämistä tiedoista ja palauttaa sen asiakassovellukselle. Kuljetusolio voi olla yksinkertainen olio tai komposiittiolio, jolla on aliolioita, riippuen siitä mitä tietoja asiakas pyytää. Komposiittientiteetti toteutetaan paikallisena entiteettipapuna. Tutkielmaan liittyvässä Kurssinhallintasovelluksessa strategiaa voisi käyttää siten että Komposiittientiteetti muodostaa Kuljetusolion kurssin, opettajan ja opiskelijoiden tiedoista.



Kuva 3.26: Composite Transfer Object Strategy (Alur & al., 2003)

3.2.7 Kuljetusolio (Transfer Object)

Serialisoitavan Kuljetusolion avulla Alurin & al. (2003) mukaan siirretään useita tietoelementtejä kerrosten välillä. Tietoelementit tallennetaan samaan tietorakenteeseen ja siirretään kerralla palvelupyynnön tai –vastauksen vaatimuksesta. Fowler (2002) käyttää vastaavaa suunnittelumallia nimeltä Tiedonkuljetusolio (Data Transfer Object), joka on myös tarkoitettu siirtämään useita tietoelementtejä yhdellä metodikutsulla etäoliolle. Chen ja MacDonald

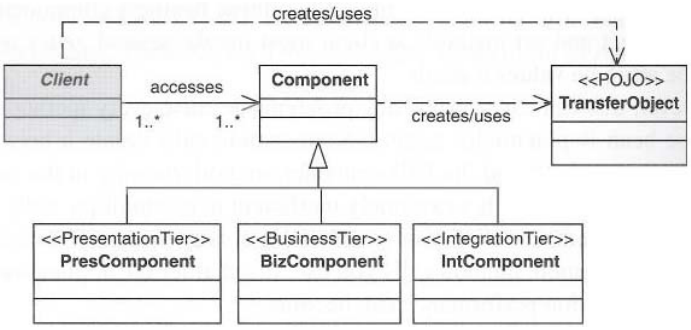
(2005) käyttävät Arvolistankäsittelijää kehittämässään työkalussa, jolla luodaan J2EE-sovellukseen arkkitehtoninen kehys suunnittelumallien avulla.

Alurin & al. (2003) mukaan Kuljetusoliota käytetään, kun

- halutaan siirtää tietoa kerrosten välillä
- halutaan vähentää etäpalvelupyyntöjä tietoliikenneverkossa
- halutaan välttää tietoliikenneverkon suorituskyvyn heikentymistä, joka johtuu sovelluksista, joissa on paljon sisäistä viestiliikennettä.

Client (kuva 3.27) tarvitsee pääsyn Component-oliolle lähettääkseen ja vastaanottaakseen tietoa. Tyypillisesti Client on jonkin toisen kerroksen Component-olio. Esimerkiksi esityskerroksen Component-olio voi esiintyä asiakassovelluksena liiketoimintakerroksen komponenteille.

Component voi olla mikä tahansa komponentti jossain toisessa kerroksessa, mihin asiakassovellus ottaa yhteyttä lähettääkseen ja vastaanottaakseen tietoa. Komponentti voi olla esityskerroksessa (PresComponent), liiketoimintakerroksessa (BizComponent) tai integrointikerroksessa (IntComponent).



Kuva 3.27: Kuljetusolio (Alur & al., 2003)

TransferObject (Kuljetusolio) on tavallinen, serialisoitava Java-olio, jolla yhdistetään ja siirretään tietoa yhden metodin avulla. TransferObject muodostetaan Component-olion vaatimuksesta. TransferObject serialisoidaan asiakassovellukselle ts. Component-olio muodostaa TransferObject-oliosta kopion, joka lähetetään Client-oliolle. Client vastaanottaa serialisoidun TransferObject-olion, jota Client käyttää paikallisena kopionaan. Client voi myös luoda TransferObject-olion ja lähettää sen Component-oliolle suorittaakseen päivittämisen.

Taulukossa 3.15 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

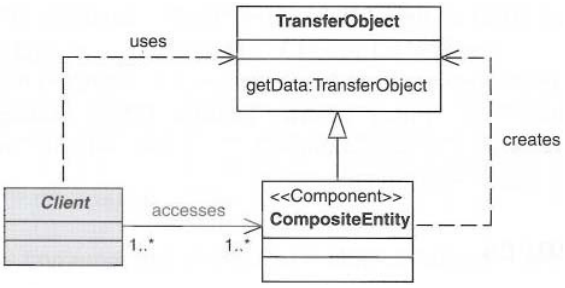
Taulukko 3.17: Kuljetusolion vaikutuksia.

Vaikutus	Selitys
Vähentää tietoliikenneverkon kuormitusta	Kuljetusolio siirtää joukon tietoja etäoliolta (esitys-, liiketoiminta- tai integrointikerroksen komponentilta) asiakassovellukselle yhden metodikutsun avulla vähentäen siten etäkutsujen määrää.
Yksinkertaistaa etäoliota ja etärajapintaa	Etäoliot (esitys-, liiketoiminta- tai integrointikerroksen komponentit) tarjoaa karkearakenteiset getData()- ja setData()-metodit tietojen arvojen asettamiseksi. Karkearakenteisten metodien avulla eliminoidaan hienorakenteiset get()- ja set()-metodit etäolioissa.
Vähentää koodin kopioitumista	Entity Inherits Transfer Object -strategian avulla vähennetään tai eliminoidaan koodin kopiointia kuljetusolion ja entiteetin välillä.
Tieto voi vanhentua kuljetusolioissa	Tieto voi vanhentua asiakassovelluksessa esim. jonkin toisen tahon suorittaman päivittämisen jälkeen. Tämä on yleinen sivuvaikutus, koska etäoliot eivät tarkkaile onko jokin asiakassovellus hakenut saman tiedon aiemmin.

Kuljetusolion toteutuksesta esitellään Alurin & al. (2003) kuvaama strategia, Entity Inherits Transfer Object Strategy. Strategia on valittu esitettäväksi tutkielmassa, koska se mahdollistaa EJB-teknologian käyttämisen suunnittelumallin kanssa.

Entity Inherits Transfer Object Strategy

Toteutettaessa Component-luokka (kuva 3.28) entiteettipapuna, asiakassovelluksen tarvitsema tieto on komponentissa itsessään. Koska kyseessä on yksi-yhteen suhde entiteetti-pavun ja sen kuljetusolion välillä, entiteetti-papu voi periä kuljetusolion. Entiteetti-pavulla on siten samat set()- ja get()-metodit kuin kuljetusoliolla. Kuljetusolioita voi olla useita erilaisia riippuen siitä minkälaisia tietokokonaisuuksia entiteetti-pavusta halutaan muodostaa. Yhden tyyppinen kuljetusolio on siten osajoukko entiteetti-pavun sisältämästä tietojoukosta. Kutakin kuljetusoliota varten on omat getData()- ja setData()-metodinsa, joita asiakassovellus kutsuu saadakseen tai päivittääkseen Kuljetusolion instanssin. Strategia vähentää koodin kopioitumista entiteetti-pavun ja kuljetusolion välillä, mutta toisaalta muutokset kuljetusolioon saattavat vaikuttaa kaikkiin sen aliluokkiin.



Kuva 3.28: Entity Inherits Transfer Object Strategy (Alur & al., 2003)

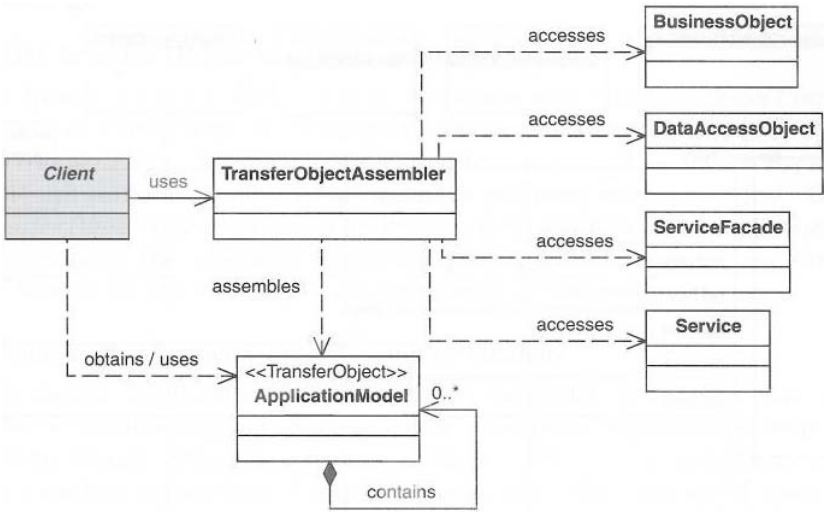
TransferObject (kuva 3.28) toteuttaa yhden tai useamman getData()- ja setData()-metodin, jotka entiteetti-papu perii. Asiakassovellus kutsuu perittyä getData()-metodia saadakseen TransferObject-instanssin. Tutkielmaan liittyvässä Kurssinhallintasovelluksessa strategiaa voisi käyttää siten, että esimerkiksi kurssin, opettajan ja opiskelijoiden tiedoista muodostetaan Kuljetusolio.

3.2.8 Kuljetusolioiden kokooja (Transfer Object Assembler)

Kuljetusolioiden kokooja tuottaa Alurin & al. (2003) mukaan sovellusmallin yhdistämällä useita Kuljetusolioita erilaisista liiketoimintakomponenteista ja –palveluista komposiitiksi Kuljetusolioksi. Kuljetusolioiden kokooja noutaa Kuljetusoliot liiketoimintakomponenteilta, kuten Liiketoimintaolioilta, Sessiojulkisivuilta, Sovelluspalveluilta ja Tiedonhakuolioilta. Kuljetusolioiden kokooja voi käyttää Palvelunpaikallistajaa paikallistaakseen liiketoiminta-palvelukomponentin, kuten Sessiojulkisivun. Asiakassovellus käyttää Kuljetusolioiden kokoojaa ”vain luku”-tarkoituksiin esittääkseen sovellusmallin tai suorittaakseen jotain muuta väliprosessointia. Liiketoimintaolio voi myös tuottaa komposiitin Kuljetusolion kapseloimastaan tiedosta. Chen ja MacDonald (2005) käyttävät Kuljetusolioiden kokoojaa kehittämässään työkalussa, jolla luodaan J2EE-sovellukseen arkkitehtoninen kehys suunnittelumallien avulla.

Alurin & al. (2003) mukaan Kuljetusolioiden kokoojaa käytetään, kun

- halutaan keskittää ja kapseloida liiketoimintalogiikka
- halutaan minimoida kutsujen määrä liiketoimintakerroksen etäolioille sovellusmallin luomiseksi
- halutaan luoda monimutkainen malli asiakassovellukselle esittämistarkoitukseen.



Kuva 3.29: Kuljetusolioiden kokooja (Alur & al., 2003)

Client (kuva 3.29) kutsuu TranserObjectAssembleria saadakseen sovellusmallin tarvitseman tiedon. Client voi olla esityskerroksen komponetti tai Sessiojulkisivu, joka tarjoaa etäyhteyskerroksen asiakassovellukselle Kuljetusolioiden kokoojan käyttöä varten. Jos Kuljetusolioiden kokooja on toteutettu sessiopapuna, Client voi olla Sessiojulkisivu tai Liiketoimintadelegaatti.

TransferObjectAssembler rakentaa komposiitin kuljetusolion asiakassovelluksen vaatimusten mukaan, kun asiakassovellus tekee palvelupyynnön sovellusmallin saamiseksi. ApplicationModel (sovellusmalli) on asiakassovellukselle palautettava komposiitti Kuljetusolio, jonka TransferObjectAssembler rakentaa.

BusinessObject on Liiketoimintaolio, joka tuottaa Kuljetusolioita kokoojalle (TransferObjectAssembler) ApplicationModel-olion koostamiseksi. ServiceFacade on Sessiojulkisivu, joka tuottaa osan ApplicationModel-kuljetusolion rakentamiseen tarvittavasta tiedosta. DataAccessObject on Tiedonhakuolio, jota TransferObjectAssembler tarvitsee hakeakseen tietoa suoraan pysyvästä tietolähteestä. Service on mielivaltainen palveluolio (esim. Sovelluspalvelu) liiketoimintakerroksessa, joka tuottaa ApplicationModel-olion rakentamisessa tarvittavaa tietoa.

Taulukossa 3.16 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.18: Kuljetusolion kokoojan vaikutuksia.

Vaikutus	Selitys
Erottaa liiketoimintalogiikan ja yksinkertaistaa asiakassovelluksen logiikkaa	Kuljetusolioiden kokooja sisältää liiketoimintalogiikan sen sisältämien olioiden suhteiden määrittämiseksi ja sovellusmallia esittävän komposiitin kuljetusolion rakentamiseksi. Asiakassovelluksen ei tarvitse tietää miten ja minkä komponenttien avulla sovellusmalli rakennetaan.
Vähentää asiakassovelluksen ja sovellusmallin kytkentää	Kuljetusolioiden kokooja piilottaa sovellusmallin rakentamisen kompleksisuuden asiakassovelluksilta ja vähentää siten kytkentää asiakassovelluksen ja sovellusmallin välillä. Malliin tarvittavat muutokset koskevat vain Kuljetusolioiden kokoojaa, eikä asiakassovellukseen tarvitse tehdä vastaavia muutoksia.
Parantaa tietoliikenneverkon suorituskykyä	Kuljetusolioiden kokooja vähentää liiketoimintakerrokseen kohdistuvia sovellusmallin rakentamiseen tarvittavia etäpyyntöjen lukumäärää, koska sovellusmalli saadaan rakennettua yhden metodikutsun avulla. Toisaalta palautettavan tiedon määrä voi olla suuri.
Parantaa asiakassovelluksen suorituskykyä	Palvelimella suoritettava Kuljetusolioiden kokooja rakentaa sovellusmallin, komposiitin Kuljetusolion, ilman asiakassovelluksen resursseja.
Kuljetusolioiden kokoojan tuottama tieto voi olla vanhentunutta	Kuljetusolioiden kokooja rakentaa sovellusmallin sillä hetkellä saatavilla olevasta tiedosta. Kun sovellusmalli on palautettu asiakassovellukselle, mahdolliset muutokset samoihin liiketoimintakompponenttien tietoihin eivät päivity asiakassovelluksella olevaan sovellusmallin tietoihin.

Kuljetusolioiden kokoojan toteutuksesta esitellään Alurin & al. (2003) kuvaama strategia, Session Bean Transfer Object Strategy. Strategia on valittu esitettäväksi tutkielmassa, koska se mahdollistaa EJB-teknologian käyttämisen suunnittelumallin kanssa.

Session Bean Transfer Object Assembler Strategy

Strategiassa Kuljetusolion kokooja toteutetaan tilattomana sessiopapuna liiketoimintapalveluksi. Sovellusmallin tuottamisessa mukana olevat liiketoimintakomponentit ovat jatkuvassa vuorovaikutuksessa monien asiakassovelluksien kanssa. Kuljetusolion kokoojan rakentaessa uuden komposiitin kuljetusolion, se sisältää tilanteen sillä hetkellä saatavilla olevasta tiedosta. Tieto voi olla vanhentunutta heti rakentamishetken jälkeen. Tilallinen sessiopapu, joka pitäisi tallessa tietoa, olisi tässä tapauksessa turhaa, koska tieto olisi todennäköisesti vanhentunutta. Kuljetusolion kokooja kannattaa siten toteuttaa tilattomana sessiopapuna. Kuljetusolion kokooja voidaan toteuttaa myös tilallisena, jolloin tulee myös rakentaa mekanismi, jolla muutokset sovellusmallin sisältämään tietoon havaitaan. Jos sovellusmallia koskeva tieto on muuttunut kun asiakassovellus tekee palvelupyynnön sovellusmallin saamiseksi, rakennetaan uusi sovellusmalli sillä hetkellä saatavilla olevasta tiedosta. Tutkielmaan liittyvässä

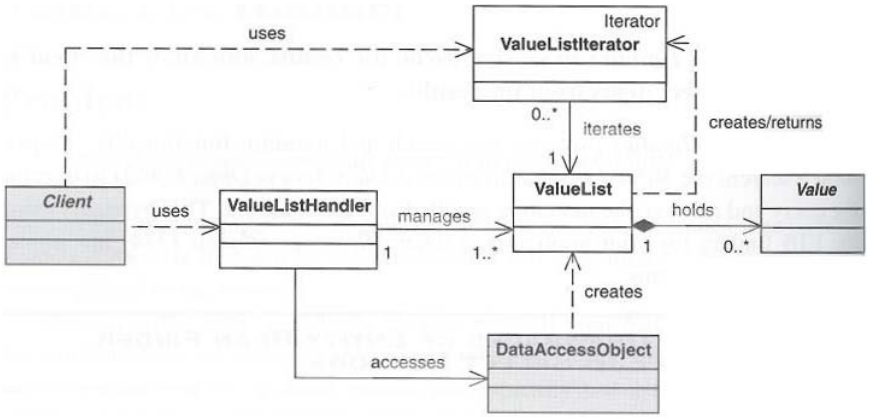
Kurssinhallintasovelluksessa strategiaa voisi käyttää siten että Komposiittientiteetti muodostaa kokoojan avulla Kuljetusolion kurssin, opettajan ja opiskelijoiden tiedoista.

3.2.9 Arvolistankäsittelijä (Value List Handler)

Arvolistankäsittelijän avulla Alurin & al. (2003) mukaan haetaan tietoja välimuistiin, sallitaan asiakassovelluksen liikkua tulosjoukon esiintymissä ja valita esiintymiä tarkasteltavaksi. Tietojen hakemiseen käytetään Tiedonhakuoliota. Tiedonhakuolio suorittaa kyselyn ja palauttaa haun tulokset tietokannasta. Tulokset tallennetaan lista-tyyppiseen tietorakenteeseen säilytettäväksi palvelimen puolella. Asiakassovellus saa listasta vain osan kerrallaan. Chen ja MacDonald (2005) käyttävät Arvolistankäsittelijää kehittämässään työkalussa, jolla luodaan J2EE-sovellukseen arkkitehtoninen kehys suunnittelumallien avulla.

Alurin & al. (2003) mukaan Arvolistankäsittelijää käytetään, kun

- halutaan välttää käyttämästä EJB hakumetodeja laajan tulosjoukon tuottaviin kyselyihin
- halutaan toteuttaa ”vain luku” käyttötapaus, joka ei edellytä tietojen vaihtoa
- asiakassovellukseen halutaan toteuttaa tehokas tiedonhaku- ja iterointimekanismi laajan tulosjoukon tuottaviin kyselyihin
- haun tulosjoukon säilyttäminen halutaan tapahtuvan palvelimella.



Kuva 3.30: Arvolistankäsittelijä (Alur & al., 2003)

Client (kuva 3.30) on asiakassovellus, joka haluaa tuottaa laajan tulosjoukon tuottavan kyselyn. Asiakassovellus voi olla esityskerroksen komponentti, joka esittää tulosjoukon käyttäjälle. Asiakassovellus voi olla myös sessiopapu, joka kapseloi Arvolistankäsittelijän.

ValueListIterator tuottaa iterointimekanismin ValueList-olion sisällölle. ValueListHandler suorittaa kyselyn ja hallinnoi tulosjoukkoa, jota se säilyttää yksityisessä ValueList-kokoelmassa. ValueListHandler luo ja manipuloi ValueList-kokoelman Tiedonhakuolion avulla. Kun asiakassovellus tekee palvelupyynnön, ValueListHandler luo ValueList-kokoelmasta osajoukon ja palauttaa sen asiakassovellukselle. Tavallisesti ValueListHandler sisältää vain yhden ValueList-olion, mutta se voi myös sisältää niitä useita, mikäli sen täytyy yhdistellä ja käsitellä useita hakutuloksia. ValueListHandler käyttää DataAccessObject-oliota tietolähteeseen päästäkseen, suorittaakseen kyselyn ja saadakseen tulosjoukon. ValueList on lista-tyyppinen kokoelma, joka säilyttää kyselyn tulokset. Value on yksittäinen kyselyn tulos-olio.

Taulukossa 3.17 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

Taulukko 3.19: Arvolistankäsittelijän vaikutuksia.

Vaikutus	Selitys
Tehokas vaihtoehto EJB-hakumetodeille	Arvolistankäsittelijä on vaihtoehtoinen tapa toteuttaa hakutoiminnot ja välttää soveltumattomien EJB-hakumetodien käyttö laajan tulosjoukon tuottavissa kyselyissä.
Kopioi välimuistiin hakutulokset	Tulosjoukko on syytä kopioida välimuistiin, kun asiakassovellus näyttää kerrallaan vain osan tuloksista. Tulosjoukko voi olla kokoelma iteroitavia kuljetusolioita.
Joustavat hakuominaisuudet	Arvolistankäsittelijän avulla voi toteuttaa älykkäät haku- ja välimuistialgoritmit, toisin kuin EJB-hakumetodien avulla.
Tehostaa tietoliikenneverkon suorituskykyä	Tietoliikenneverkon suorituskyky tehostuu, koska asiakassovellukselle lähetetään vain sen kulloinkin tarvitsema osa laajasta tulosjoukosta. Koko tulosjoukkoa säilytetään palvelimen puolella. Asiakassovellukselle voidaan tarvittaessa lähettää koko tulosjoukko kerralla erillisen metodin avulla.
Entiteettipavun muokkaamiseen käytettävä tiedonvaihto suoritetaan vain tarvittaessa	Tiedon muokkaamiseen käytetään entiteettipavun hakumetodeja, jotka palauttavat muokattavan entiteettipavun. Arvolistankäsittelijä on ”vain luku”-tyyppinen, joten muokkaamisen suorittamiseksi käytetään tarvittaessa entiteettipavun hakumetodeja. Kun käyttäjä haluaa muokata arvolistassa olevan entiteetin arvoja, käytetään entiteettipavun hakumetodeja, jotka palauttavat muokattavan entiteettipavun.
Edistää kerrostamisen ja erottelun ajattelutapaa	Arvolistankäsittelijä kapseloi tulosjoukon sisältävän listan käsittelyn liiketoimintakerrokseen ja käyttää tarkoituksenmukaisesti Tiedonhakuoliota integrointikerroksessa. Siten liiketoimintalogiikka pidetään liiketoimintakerroksessa ja tiedonhakulogiikka integrointikerroksessa.
Laajan listan luominen voi olla kallista	Kun Tiedonhakuolio suorittaa kyselyn ja luo kokoelman Kuljetusolioita, se kuluttaa merkittävästi resursseja jos tulosjoukko on suuri. Sen sijaan että koko tulosjoukosta luotaisiin Kuljetusolioita, voidaan Tiedonhakuolion palauttamien tuloksien määrää rajoittaa asettamalla maksimäärä, jonka tietokannasta voi noutaa.

Arvolistankäsittelijän toteutuksesta esitellään Alurin & al. (2003) kuvaama strategia, ValueList from Data Access Object Strategy. Strategia on valittu esitettäväksi tutkielmassa, koska se yhdistää Tiedonhakuolio- ja Arvolistankäsittelijä –suunnittelumallit ja mahdollistaa Java-papujen tehokkaan käytön käsiteltäessä laajaa tulosjoukkoa.

ValueList from Data Access Object strategy

Strategiassa Arvolista saadaan Tiedonhakuolion avulla käyttäen RowSet Wrapper List -strategiaa (ks. Tiedonhakuolio), jota voidaan käyttää laajan tulosjoukon tuottavan kyselyn suorittamiseksi.

3.3 Integrointikerroksen suunnittelumalleja

Tässä luvussa esitellään J2EE-arkkitehtuurin integrointikerroksen suunnittelumalleja. Mukana on kuvan 1.1 integrointikerroksen suunnittelumalleista Tiedonhakuolio (Data Access Object). Suunnittelumallista on kerrottu kuvaus, käyttötarkoitus, vaikutukset ja toteutusstrategioita. Tutkielmaan liittyvässä Kurssinhallintasovelluksessa ei ole käytetty integrointikerroksen suunnittelumalleja, mutta Tiedonhakuolion strategiassa pohditaan soveltamista.

3.3.1 Tiedonhakuolio (Data Access Object)

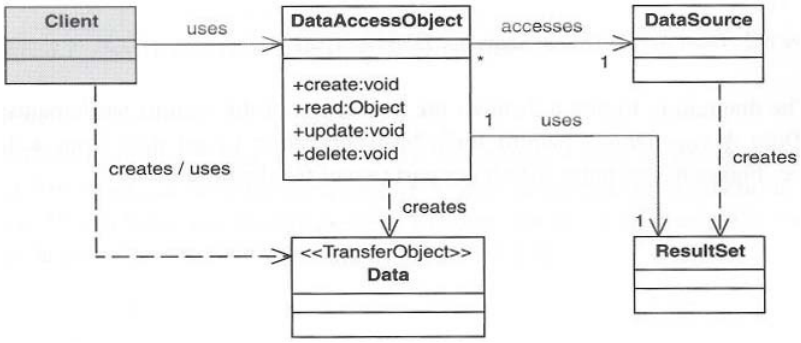
Tiedonhakuolio abstrahoi ja kapseloi kaikki pääsyt pysyvään tietolähteeseen. Alurin & al. (2003) mukaan Tiedonhakuolio hoitaa yhteyden tietolähteeseen, hakemista ja tallentamista varten. Tiedonhakuolio tarjoaa yhtenäisen tiedonhaku API:n asiakassovellukselle ja piilottaa tietolähteen toteutuksen yksityiskohdat asiakassovelluksilta. Liiketoimintakomponentti, joka tarvitsee pääsyn tietolähteeseen, käyttää Tiedonhakuolion tarjoamaa yksinkertaista rajapintaa. Tietolähteen toteutusta voi muuttaa asiakasrajapintaa muuttamatta. Tiedonhakuolio toimii sovittimena asiakassovelluksen ja tietolähteen välillä. Nock (2004) kertoo tarkemmin tietokantaohjelmointiin liittyvistä tekniikoista, joiden avulla esim. sivuttamiseen ja välimuistiin liittyvät ongelmat ratkaistaan. Dragomiroiu & al. (2003) sijoittavat Tiedonhakuolion mallikerrokseen MVC-mallissa. Hammouda ja Koskimies (2002) sekä Chen ja MacDonald (2005) ovat kehittäneet työkalut, joilla luodaan sovelluksen arkkitehtoninen kehys suunnittelumallien avulla. Molemmissa työkaluissa on käytetty Tiedonhakuoliota.

Alurin & al. (2003) mukaan Tiedonhakuoliota käytetään, kun

- halutaan luoda mekanismi, jonka avulla luodaan yhteys tietolähteeseen ja manipuloidaan pysyvästi tallennettavaa tietoa
- halutaan erottaa pysyvästi tallennettava tieto muusta sovelluksesta

- halutaan luoda yhtenäinen API pysyvästi tallennettavalle tiedolle eri tyyppisille tietolähteille, kuten RDBMS (Relational database management systems), LDAP (Lightweight Directory Access Protocol), OODB (object-oriented databases), XML- (Extensible Markup Language) tietovarastot, tiedostot jne.
- Halutaan organisoida logiikka jolla hallitaan tiedon hakeminen tietolähteestä ja kapseloidaan ylläpidettävyys- ja siirrettävyys-ominaisuuksia.

Tiedonhakuolio toteutetaan tilattomana oliona. Se ei säilö kyselyn tulosjoukkoa tai mitään muutakaan tietoa, mitä asiakassovellus voisi tarvita myöhemmin uudelleen. Tämän vuoksi Tiedonhakuolio on kevytrakenteinen olio ja välttää potentiaaliset säie- ja samanaikaisuusongelmat. Tiedonhakuolio kapseloi kaikki JDBC- (Java Database Connectivity) käsittelyt tiedonhakukerrokseen, paljastamatta kerroksen ulkopuolisille asiakassovellukselle poikkeuskäsittelyä, tietorakennetta tai rajapintoja, jotka kuuluvat java.sql.*- tai javax.sql.*-paketteihin.



Kuva 3.31: Tiedonhakuolio (Alur & al., 2003)

Client (kuva 3.31) on olio, joka vaatii pääsyä tietolähteeseen hakeakseen ja tallentaakseen tietoa. Se voi olla Liiketoimintaolio, Sessiojulkisivu, Sovelluspalvelu, Arvolistankäsittelijä tai jokin muu avustava olio, joka tarvitsee pääsyn pysyvään tietolähteeseen.

DataAccessObject abstrahoi tiedonhaun toteutuksen Client-oliota varten mahdollistaakseen pääsyn tietolähteeseen. DataAccessObject toteuttaa lisäys-, haku-, päivitys- ja poisto-operaatiot.

DataSource edustaa tietolähteen toteutusta. Tietolähde voi olla tietokanta, kuten RDBMS, OODBMS, XML tietovarasto, tiedosto jne, jokin toinen järjestelmä (B2B palvelu tai luottokorttitoimisto) tai esim. LDAP tietovarasto.

ResultSet edustaa kyselyn tulosjoukkoa. Kun tietolähteenä on RDBMS, sovelluksen käyttäessä JDBC API:a, ResultSet on java.sql.ResultSet-olio.

Data edustaa kuljetusoliota jota käytetään tiedon siirtämiseen. DataAccessObject voi käyttää Kuljetusoliota tiedon palauttamiseksi asiakassovellukselle. DataAccessObject voi myös saada tiedon Kuljetusoliona päivitettäväksi tietolähteeseen.

Taulukossa 3.18 on kuvattu mallin vaikutuksia Alurin & al. (2003) perusteella.

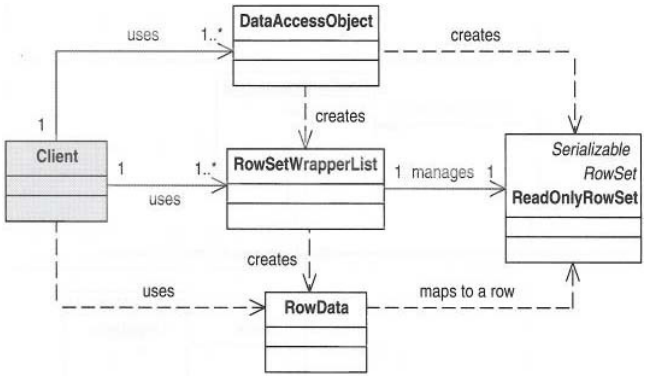
Taulukko 3.20: Tiedonhakuolion vaikutuksia.

Vaikutus	Selitys
Tuottaa oliosuuntautuneen näkymän ja kapseloi tietokantaskeemat	Asiakassovellukset käyttävät kuljetusolioita tai kursoriolioita tiedon vaihtamiseen Tiedonhakuolioiden kanssa. Sen sijaan että asiakassovellukset olisivat riippuvaisia alemman kerroksen tietokantaskeeman toteutuksista, kuten ResultSet- tai RowSet-olioista, joita käyttäessä tulee olla tietoinen taulurakenteista, sarakkeiden nimistä jne., asiakkaat käsittelevät tietoa oliosuuntautuneesti käyttäen kuljetusolioita ja kursoriolioita.
Helpottaa tietolähteen muuttamista	Asiakassovelluksilla ei ole tietoa allaolevasta tietolähteen toteutuksesta, joten tietolähteeseen kohdistuvat muutokset koskevat ainoastaan Tiedonhakuoliota. Tiedonhakuolio toimii kuten Välittäjä vähentäessään kytkentää ylempien kerrosten ja integrointikerroksen välillä.
Vähentää asiakassovellusten kompleksisuutta	Tiedonhakuolio kapseloi toiminnot vuorovaikutukseen tietolähteen kanssa, jolloin asiakassovellukset voivat käyttää yksinkertaista API:a tiedon tallentamiseen tai hakemiseen. Tämä vähentää asiakassovelluksen koodin kompleksisuutta.
Tuottaa ylimääräisen kerroksen	Tiedonhakuolio luo lisäkerroksen asiakkassovelluksen ja tietolähteen väliin. Lisäkerroksen tekeminen vaatii suunnittelu- ja toteutusresursseja, mutta on tavallisesti välttämätöntä erottaa pysyvän tietolähteen käsittelyn toteutus muusta sovelluksesta.

Tiedonhakuolion toteutuksesta esitellään Alurin & al. (2003) kuvaama strategia, RowSet Wrapper List Strategy. Strategia on valittu esitettäväksi tutkielmassa, koska sen avulla voidaan yhdistää Tiedonhakuolio- ja Arvolistankäsittelijä –suunnittelumallit ja se mahdollistaa Java-papujen tehokkaan käytön käsiteltäessä laajaa tulosjoukkoa.

RowSet Wrapper List Strategy

Strategia soveltuu suuren tulosjoukon käsittelyyn. Käärityn RowSet-instanssi paljastaa kyselyn tulosjoukon oliosuuntautuneesti. Kääriminen tehdään kustomoidun listan avulla, joka tuottaa listassa liikkumiseen ja varastoimiseen tarvittavat funktiot. RowSet-oliota manipuloidaan olisuuntautuneesti paljastamalla tietueen sisältämän tieto olioina. Kuljetusolioita luodaan ja palautetaan asiakassovellukselle vain tarvittaessa.



Kuva 3.32: RowSet Wrapper List Strategy

DataAccessObject (kuva 3.32) luo ReadOnlyRowSet-olion ja välittää sen parametrina luotavalle RowSetWrapperList-oliolle. DataAccessObject palauttaa RowSetWrapperList-olion Client-oliolle, joka käyttää sen listaa käydäkseen läpi kyselyn tulosjoukkoa. RowSetWrapperList toteuttaa List-rajapinnan tuottaakseen listassa liikkumiseen tarvittavat funktiot ReadOnlyRowSet-oliossa olevaa tietoa varten. RowSetWrapperList kapseloi kaikki tiedonhakukerroksen toiminnallisuuden, jolloin Client-olion ei tarvitse tietää mitään RowSet-, ResultSet- tai minkään muun java.sql- tai javax.sql-paketin luokista. RowSetWrapperList palauttaa Kuljetusolion silloin, kun asiakas haluaa edellisen tai seuraavan tietueen tulosjoukosta. Tutkielmaan liittyvässä Kurssinhallintasovelluksessa strategiaa voisi käyttää siten, että Tiedonhakuolio hakee kurssin, opettajan ja opiskelijoiden tiedot ja muodostaa niistä Kuljetusolion, joka palautetaan asiakkaalle. Asiakassovelluksessa käyttäjä voi selata kurssin opiskelijatietoja. Kuljetusoliossa on kerrallaan vain osa tiedoista ja Tiedonhakuolio hakee tarvittaessa lisää opiskelijoita tietokannasta ja palauttaa ne asiakkaalle Kuljetusoliossa.

4 YHTEENVETO

Suunnittelumallit antavat yhteisen sanaston, jota voidaan käyttää keskusteluissa ja dokumentoinnissa. Kun sekä puhuja että kuuntelijat tuntevat yleisimpiä suunnittelumalleja, voidaan sanoa, että ”tässä käytetään Abstraktia tehdasta” sen sijaan, että kuvaillaan monimutkaista mekanismia, jolla synnytetään olioita tuntematta niiden konkreettisia luokkia. Yhteinen sanasto nostaa käytävän keskustelun abstraktiotasoa.

Suunnittelumallien opiskelu helpottaa järjestelmien suunnittelua ja olemassa olevien järjestelmien ymmärtämistä. Aloittelija voi mallien avulla oppia käyttämään ratkaisuja, joita kokeneet suunnittelijat tekevät. Toivosen (2001) mukaan suunnittelumallit ovat keskeinen osa suunnittelumenetelmiä. Koskimiehen (2000) mukaan suunnittelumallit ovat tapa, jolla ohjelmiston kehittäjien hyväksi koetut ratkaisut voidaan tallettaa siten, että niiden tuntemus säilyy, vaikka henkilöt vaihtuvat.

Suunnittelumallit oikein käytettynä parantavat mm. ohjelmiston uudelleenkäytettävyyttä, laajennettavuutta ja korvattavuutta. Haittavaikutuksena voi olla mm. aikavaativuuden kasvaminen, koska esim. uudelleenkäytettävyyden huomioiminen vaatii yleisen ratkaisun, joka on usein tapauskohtaista ratkaisua monimutkaisempi ja joskus hieman tehottomampi. Suunnittelumallit väärinkäytettynä voivat pikemminkin haitata ohjelmiston kehittämistä kuin parantaa sitä. Järjestelmä ei välttämättä ole laadukas, jos sanotaan, että siinä on käytetty runsaasti suunnittelumalleja. Koskimiehen (2000) mukaan suunnittelumallin soveltaminen tulee lähteä aina ongelman identifioimisesta.

J2EE-suunnittelumallit ovat kehittyneet J2EE-teknologian päälle toteutettujen järjestelmien suunnittelutyössä. J2EE-arkkitehtuuriin kuuluvat mm. EJB-, XML-, SOAP-, JMS-, JSTL- ja servlet-teknologiat niitä tukevan sovelluspalvelimen (Apache Tomcat, Orion Server, Jboss, BEA WebLogic, IBM WebSphere, jne.) päälle rakennettuna. Uudet teknologiat vaativat uusia tapoja soveltaa edelleen käyttökelpoisia suunnittelumalleja. Ns. vanhat tai yleiset suunnittelumallit sopivat sellaisenaan sovellettavaksi J2EE-teknologian päälle rakennettavaan ohjelmistoon, mutta myös uusia suunnittelumalleja tarvitaan ratkaisemaan uuden teknologian myötä esiintulleita ongelmia. Alur & al. (2003) esittelevät joukon J2EE-suunnittelumalleja, jotka ovat syntyneet J2EE-teknologiaa hyväksikäyttävien ohjelmistojen suunnittelijoiden työstä.

Nämä suunnittelumallit, kuten ns. yleiset suunnittelumallitkin, on siten yleisesti hyväksi koettuja ratkaisuja usein esille tulevaan ongelmaan. J2EE-suunnittelumallit voivat käyttää hyväksi yleisiä suunnittelumalleja osana tietyn ongelman ratkaisevassa suunnittelumallissa.

J2EE-suunnittelumallien hyviä ja huonoja puolia on koottu taulukoihin 4.1 ja 4.2. Taulukoissa on allekain Alurin & al. (2003) esittelemiä suunnittelumalleja ja sarakkeissa hyviä (taulukko 4.1) ja huonoja (taulukko 4.2) puolia. Hyvät ja huonot ominaisuudet on poimittu luvussa 3 taulukoiduista vaikutuksista. Ominaisuuksia on yhdistetty samojen otsikkojen alle, esim. ”yksinkertaistaa asiakassovelluksen logiikkaa” on laitettu taulukossa sarakkeeseen ”vähentää kompleksisuutta”.

Taulukko 4.1. Suunnittelumallien hyvät puolet.

	Keskittäminen	Uudelleenkäytettävyys	Ylläpidettävyy	Laajennettavuus	Mukautuvuus	Saatavuus	Suunnittelijoiden roolien erottelu	Osittaminen, prosessien erotteleminen	Testattavuus	Modulaarisuus	Vähentää kytkentää	Parantaa suorituskykyä	Vähentää kompleksisuutta	Vähentää koodin kopioitumista	Sovelluskehyksien ja kirjastojen hyödyntäminen
Sieppaussuodatin	x	x												x	x
Etukontrolleri	x	x			x		x								
Näkymäavustaja		x	x				x	x	x						
Sovelluskontrolleri		x		x						x					
Kontekstiolio		x	x	x					x						
Komposiittinäkymä		x	x							x					
Välittäjänäkymä		x						x							x
Palvelu työstäjälle	x	x	x				x			x					
Liiketoimintadelegaatti			x			x					X	x			
Palvelunpaikallistaja	x			x								x	x		
Sessiojulkisivu	x		x		x						X	x	x		

Taulukko 4.1. Suunnittelumallien hyvät puolet (jatk.).

	Keskittäminen	Uudelleenkäytettävyys	Ylläpidettävyy	Laajennettavuus	Mukautuvuus	Saatavuus	Suunnittelijoiden roolien erottelu	Osittaminen, prosessien erotteleminen	Testattavuus	Modulaarisuus	Vähentää kytkentää	Parantaa suorituskykyä	Vähentää kompleksisuutta	Vähentää koodin kopioitumista	Sovelluskehyksien ja kirjastojen hyödyntäminen
Sovelluspalvelu	x	x											x	x	
Liiketoimintaolio		x	x					x						x	
Komposiittientiteetti			x									x			
Kuljetusolio												x	x	x	
Kuljetusolioiden kokooja								x		x		x			
Arvolistankäsittelijä								x				x			
Tiedonhakuolio					x								x		

Taulukkoa 4.1 tarkastellessa voi todeta, että yleisimmät hyödyt J2EE-suunnittelumalleja käytettäessä ovat järjestyksessä uudelleenkäytettävyys, ylläpidettävyys, suorituskyvyn kasvu, keskittäminen ja osittaminen, prosessien erotteleminen ja kompleksisuuden väheneminen. Suorituskyvyn kasvu kolmanneksi suurimpana hyötynä johtuu mm. siitä, että järjestelmässä halutaan käyttää tiettyjä teknologioita (esim. EJB) ja suunnittelumallin tietty toteutusstrategia on suunniteltu tehostamaan tätä teknologiaa (Komposiittientiteetti ja Composite Transfer Object Strategy). Voi silti olla mahdollista, että sovellusaluekohtainen ratkaisu on tehokkaampi kuin suunnittelumallien tarjoama ratkaisu. Sovellusaluekohtainen ratkaisu voi kuitenkin kärsiä uudelleenkäytettävyydessä, ylläpidettävyydessä, laajennettavuudessa jne., missä suunnittelumallin mukainen ratkaisu on parempi. Kompleksisuuden väheneminen tarkoittaa asiakassovelluksen yksinkertaistumista. Asiakassovelluksesta otetaan sitä monimutkaistava ja usein myös sinne kuulumaton koodi pois ja sijoitetaan omaan loogiseen kokonaisuuteen.

Taulukko 4.2. Suunnittelumallien huonot puolet

	Suorituskyvyn lasku	Vähentää ylläpidettävyyttä	Heikko erottelu	Tuottaa ylimääräisen kerroksen, tason (layer)	Tiedon vanhentuminen oliossa	Koodin kopioituminen	Heikko osittaminen, prosessien erotteleminen	Kompleksisuus
Sieppaussuodatin	x							
Etukontrolleri								
Näkymäavustaja		x				x		
Sovelluskontrolleri								
Kontekstiolio	x							
Komposiittinäkymä	x	x						
Välittäjänäkymä							x	
Palvelu työstäjälle								
Liiketoimintadelegaatti				x				
Palvelunpaikallistaja								
Sessiojulkisivu								
Sovelluspalvelu				x				
Liiketoimintaolio				x			x	
Komposiittientiteetti								
Kuljetusolio					x			x
Kuljetusolioiden kokooja					x		x	
Arvolistankäsittelijä	x							
Tiedonhakuolio				x				

Taulukon 4.2 mukaan yleisimmät huonot puolet ovat suorituskyvyn lasku, ylimääräisen kerroksen tai tason tuottaminen ja heikko prosessien erotteleminen. Huonot puolet voivat johtua esimerkiksi ”väärästä” tavasta käyttää suunnittelumallia, jolloin seuraukset ovat negatiivisia.

Väärän käyttötavan aiheuttamia seurauksia ovat esim. suorituskyvyn laskeminen ja heikko prosessien erotteleminen. Toisaalta esimerkiksi Kontekstiolion luominen parantaa uudelleenkäytettävyyttä, ylläpidettävyyttä, laajennettavuutta ja testattavuutta suorituskyvyn kustannuksella.

Tutkielmaan liittyvässä Kurssinhallintasovelluksessa on käytetty hyväksi Sieppaussuodatin-, Etukontrolleri-, Kontekstiolio-, Sovelluskontrolleri-, Näkymäavustaja-, Välittäjänäkymä- ja Palvelu työstäjälle -suunnittelumalleja. Sovellus on yksinkertainen ja sen tarkoituksena on näyttää, miten suunnittelumalleja voidaan käyttää hyväksi ohjelmiston toteuttamisessa. Kaikkia J2EE-suunnittelumalleja ei tämän tutkielman puitteissa voitu toteuttaa sovelluksessa, joten sovellus rajoittuu arkkitehtuuriltaan esityskerroksen komponentteihin. Tietokannasta haetaan tietoja apuluokan avulla, joka tarjoaa yhteyden pysyvään tietolähteeseen. Suunnittelumallin käyttäminen lähtee kirjallisuuden mukaan ongelman tunnistamisesta. Pitää ensin olla ongelma, ennen kuin voi käyttää suunnittelumallin tarjoamaa ratkaisua. Sovelluksen toteuttamisessa suurin vaikeus oli keksiä, miten suunnittelumalleja voisi käyttää tarkoituksenmukaisesti. Suunnittelumallin tarjoamalle ratkaisulle piti joissain tapauksissa keksiä ongelma.

Etukontrolleri-suunnittelumallin käyttäminen oli sinänsä loogista, koska kontrolleri vastaanottaa kaikki palvelupyynnöt keskitetysti. Kontrolleri delegoi palvelupyynnön toiselle järjestelmän komponentille tai välittää suoraan näkymän generoivalle komponentille, kuten Välittäjänäkymä-suunnittelumallin Dispatcher in controller -strategiassa. Palvelupyynnön kohdistuessa osoitteeseen, jossa on etusivu, kontrolleri voi suoraan välittää näkymälle, joka generoi etusivun. Näkymän generoimisessa käytettiin Näkymäavustaja-suunnittelumallia hyväksi, jotta vältetään ohjausrakenteiden ohjelmointia suoraan JSP-tiedostoon. JSTL:n, tagitiedostojen ja kustomoitujen tagien avulla tehtiin avustajia, joilla ulkoasun määrittelevän HTML-koodin sekaan ei tarvinnut upottaa suurta määrää ohjelmointikieltä dynaamisten WWW-sivujen luomiseksi.

Tiedostojen lataus WWW-sivun kautta palvelimelle ja palvelimelta tarjosi ongelman, johon Sieppaussuodattimen tarjoama ratkaisu sopi hyvin. Tiedostojen lataus palvelimelle ja palvelimelta tapahtuu uudelleenkäytettävän tagitiedoston ohjaamana. Multipart-lomakkeen käsittely poikkeaa kuitenkin normaalin HTML-lomakkeen käsittelystä. Multipart-lomakkeen parametrien lukeminen vaatii erikseen sovelluspalvelimelle asennettavan apukirjaston ja komponen-

tin, joka ottaa tiedoston vastaan käyttäen apukirjastoa. Tiedoston lataamiseksi palvelimelle voidaan käyttää esimerkiksi servlettiä, joka lukee lomakkeen tiedot käyttäen em. apukirjastoa. Tällöin joutuisi käyttämään kahta erilaista tapaa jäsentää palvelupyyntöjen parametrejä ja kontrollerin ohjausta erikseen laadittavalle servletille tiedoston lataamiseksi palvelimelle. Sieppaussuodatin-suunnittelumallin avulla voidaan lukea parametrit sekä tavallisesta HTML-lomakkeesta että multipart-lomakkeesta ja tallettaa parametrien nimet ja niiden arvot siten, että ne voidaan käsitellä muualla sovelluksessa aina samalla tavalla.

Kurssien, opettajien ja opiskelijoiden tietojen käsittelyyn käytettiin Palvelu työstäjälle -suunnittelumallia, jossa Etukontrolleri-, Sovelluskontrolleri- ja Näkymäavustaja-suunnittelumalleja käytetään yhdessä. Esimerkiksi kurssin tietojen lataamiseksi tietokannasta ja näyttämiseksi näytöllä Etukontrolleri ottaa palvelupyynnön vastaan, delegoi kontrollin Sovelluskontrollerille, joka hakee toimintoa vastaavan komennon (avustajan). Sovelluskontrolleri suorittaa komennon, joka hakee kurssin tiedot tietokannasta ja palauttaa sopivan näkymän osoitteen. Sovelluskontrolleri välittää näkymälle, joka generoi avustajien avulla näytön WWW-selaimelle. Kontekstioliota käytetään kurssin päivittämisessä. Kurssi-kontekstiolio luodaan kontekstiotehtaan avulla. Kurssi-olioon talletetaan palvelupyynnön parametreina tulevat muutettavat kurssin tiedot ja kurssin yksilöivä tunniste, jonka avulla tietokantaan päivitetään kyseinen kurssi.

Sovelluksessa hyväksikäytettyjen suunnittelumallien soveltaminen oli loogista ja tarkoituksenmukaista tässä yksinkertaisessa sovelluksessa. Kirjallisuudesta löytyneet toteutusstrategiat auttoivat suunnittelumallien soveltamisessa ratkaisevasti. Mallien moninaisuuden vuoksi olisi järkevintä kuitenkin soveltaa J2EE-teknologiaan perustuvia suunnittelumalleja huomattavasti laajempaan rakennettavaan J2EE-arkkitehtuuriin tai huonosti laadittuun ohjelmistoon. Huonosti suunnitellussa ja toteutetussa ohjelmistossa on runsaasti ongelmia, joiden tunnistaminen voi johtaa jonkin tietyn suunnittelumallin tarjoaman ratkaisun käyttämiseen refaktorointina. Hammoudan ja Koskimiehen (2002) mukaan J2EE-suunnittelumallit voisivatkin parantaa merkittävästi hajautettujen yrityssovellusten suunnittelua ja arkkitehtuuria.

SANASTO

API (Application Programming Interface), niiden ohjelmapalvelujen ja tietorakenteiden joukko ja sen kuvaus, joka käyttäjän sovelluksella on käytettävissään (http://www.jyu.fi/library/sanasto/sanasto.html).

Arkkitehtuuri on kokonaisuus, joka muodostuu järjestelmän osista, osien keskinäisistä suhteista sekä toiminnan pääperiaatteista (http://www.tsk.fi/termitalkoot/).

Asiakas tai asiakassovellus (client) on kutsuva ohjelma tai ohjelman osa samassa tai eri järjestelmässä. Kutsuva ohjelma voi olla olio tai esim. WWW-selain.

BMP (bean-managed persistence) on mekanismi, jossa entiteettipapu huolehtii itse pysyvyyden hallinnasta (http://java.sun.com/javaee/reference/glossary/index.jsp).

CMP (container-managed persistence) on mekanismi, jossa entiteettipavun pysyvyydestä huolehtii EJB-säiliö (http://java.sun.com/javaee/reference/glossary/index.jsp).

CMR (container-managed relationships) on mekanismi, jolla kuvataan entiteettipapujen väliset suhteet.

Java 2 Platform, Enterprise Edition (J2EE). Palvelinsovellusympäristöihin tarkoitettu, yrityskäyttöön soveltuva java-kehityskieli. Koostuu joukoista palveluita, API:ja ja protokollia joiden avulla voidaan rakentaa monikerrosmallin mukaisia WWW-pohjaisia ohjelmistoja. (http://java.sun.com/javaee/reference/glossary/index.jsp).

Entiteettipapu (Enterprise Java Bean, EJB) on palvelimille suunnattu komponenttiteknologia (http://java.sun.com/developer/technicalArticles/ebeans/EJB20CMP).

Java-papu (JavaBean) on siirrettävä, alustariippumaton ja uudelleenkäytettävä komponenttiteknologia. (http://java.sun.com/products/javabeans/)

JDBC (Java Database Connectivity) on tietokantayhteyden API http://java.sun.com/javaee/reference/glossary/index.jsp).

JMS (Java Message Service) on API, jonka avulla kutsutaan operaatioita viestintäjärjestelmässä. (http://java.sun.com/javaee/reference/glossary/index.jsp).

JNDI (Java Naming and Directory Interface) on API, jonka avulla nimetään ja kutsutaan palveluita (http://java.sun.com/javaee/reference/glossary/index.jsp).

JSP (Java Server Pages) on palvelinpuolen tekniikka, jolla tuotetaan dynaamista sisältöä asiakassovellukselle, kuten esim. HTML-sivuja WWW-selaimelle (http://java.sun.com/javaee/reference/glossary/index.jsp).

JSTL (Java Server Pages Standard Tag Library) on tagi-kirjasto, joka sisältää JSP-sivujen yleisiä toimintoja esim. XML-tiedoston käsittelyyn (http://java.sun.com/javaee/reference/glossary/index.jsp).

LDAP (Lightweight Directory Access Protocol) on hakemistopalvelujen käyttöön tarkoitettu verkkoprotokolla (http://fi.wikipedia.org/wiki/LDAP).

Luokka kertoo, mitä attribuutteja ja operaatioita sen ilmentymällä eli oliolla on. Luokka on aihio tai muotti, jonka avulla oliota luodaan. (Koskimies, 1998).

OODB (object-oriented databases) on oliopohjainen tietokanta.

Olio sisältää ominaisuuksia eli attribuutteja ja operaatioita eli metodeja. Attribuuttien yhdistelmää kutsutaan olion tilaksi. Tilaa voi muuttaa muuttamalla attribuuttien arvoja. Attribuuttien arvoja muutetaan metodien avulla. Tiedon ja ja siihen liittyvän toiminnallisuuden pakkaaminen kokonaisuudeksi on kapselointia. (Koskimies, 1998).

Palvelupyyntö (request) tulee asiakkaalta. Olio suorittaa operaation kun se saa pyynnön asiakkaalta.

POJO (Plain Old Java Object) on tavallinen Java-olio, erotuksena esimerkiksi Java-papuihin.

RDBMS (Relational database management systems) on relaatiotietokannan hallintajärjestelmä (http://www.webopedia.com/TERM/R/RDBMS.html).

Suunnittelumalli (design pattern) on yleinen, toistettavissa oleva ratkaisu usein esille tulevaan ongelmaan ohjelmistosuunnittelussa (Toivonen, 2001).

Vastaus (response). Olio vastaa asiakkaalta tulleeseen palvelupyyntöön.

Välimuisti (cache) väliaikaiseen tallennukseen tarkoitettu tietokoneen muisti, joka nopeuttaa tietojen hakemista (http://www.tsk.fi/termitalkoot/).

WSDL (Web Services Description Language) on verkkopalveluiden kuvaamiseen käytettävä XML-rakenne (http://www.w3.org/TR/wsdl).

XML- (Extensible Markup Language) on SGML-kielestä erityisesti Internet-käyttöä varten rajattu merkintäkieli, joka on helposti laajennettavissa (http://www.tsk.fi/termitalkoot/).

VIITELUETTELO

Alur D., Crupi J., Malks D.: *Core J2EE Patterns: Best Practises and Design Strategies, Second Edition*. Sun Microsystems Press, 3. painos, 2003.

Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M.: *Pattern-Oriented Software Architecture: a System of Patterns*. John Wiley & Sons, 1996.

Chen J., MacDonald S.: RoadMapAssembler: a new pattern-based J2EE development tool. *Proceedings of the 2005 conference of the Centre for Advanced Studies on Collaborative research CASCON '05*, IBM Press, Toronto, 55-69, 2005.

Dragomiroiu M., Gyorodi R., Salomie I., Gyorodi C.: Designing Reusable Web-Applications by Employing Enterprise Frameworks. *Lecture Notes in Computer Science*, vol 2869, 1051-1059, 2003.

Jacyntho M., Schwabe D., Rossi G. *A Software Architecture for Structuring Complex Web Applications*. http://www2002.org/CDROM/alternate/478/ (9.8.2006).

Fowler M.: *Patterns Of Enterprise Application Architecture*. Addison-Wesley, 2002.

Hammouda I., Koskimies K.: A Pattern-Based J2EE Application Development Environment. *Proceedings of the Nordic Workshop on Software Development Tools and Techniques* (toim. Osterbye K.), NWPER 2002, IT University of Copenhagen, 2002.

Knight A, Dai N.: Objects and the Web. *IEEE Software*, 19(2), 2002.

Koskimies K.: *Pieni oliokirja*. Suomen Atk-kustannus Oy, 2. painos, 1998.

Koskimies K.: *Oliokirja*. Satku - Kauppakaari, 2. painos, 2000.

Maciaszek L., Liong B.: *Practical Software Engineering: a case study approach*. Pearson Education Limited, 2005.

Metsker S.: *Design Patterns Java Workbook*. Addison-Wesley, 2002.

Neill C., Gill B.: Refactoring reusable business components. *IT Professional*, 5(1), 33-38, 2003.

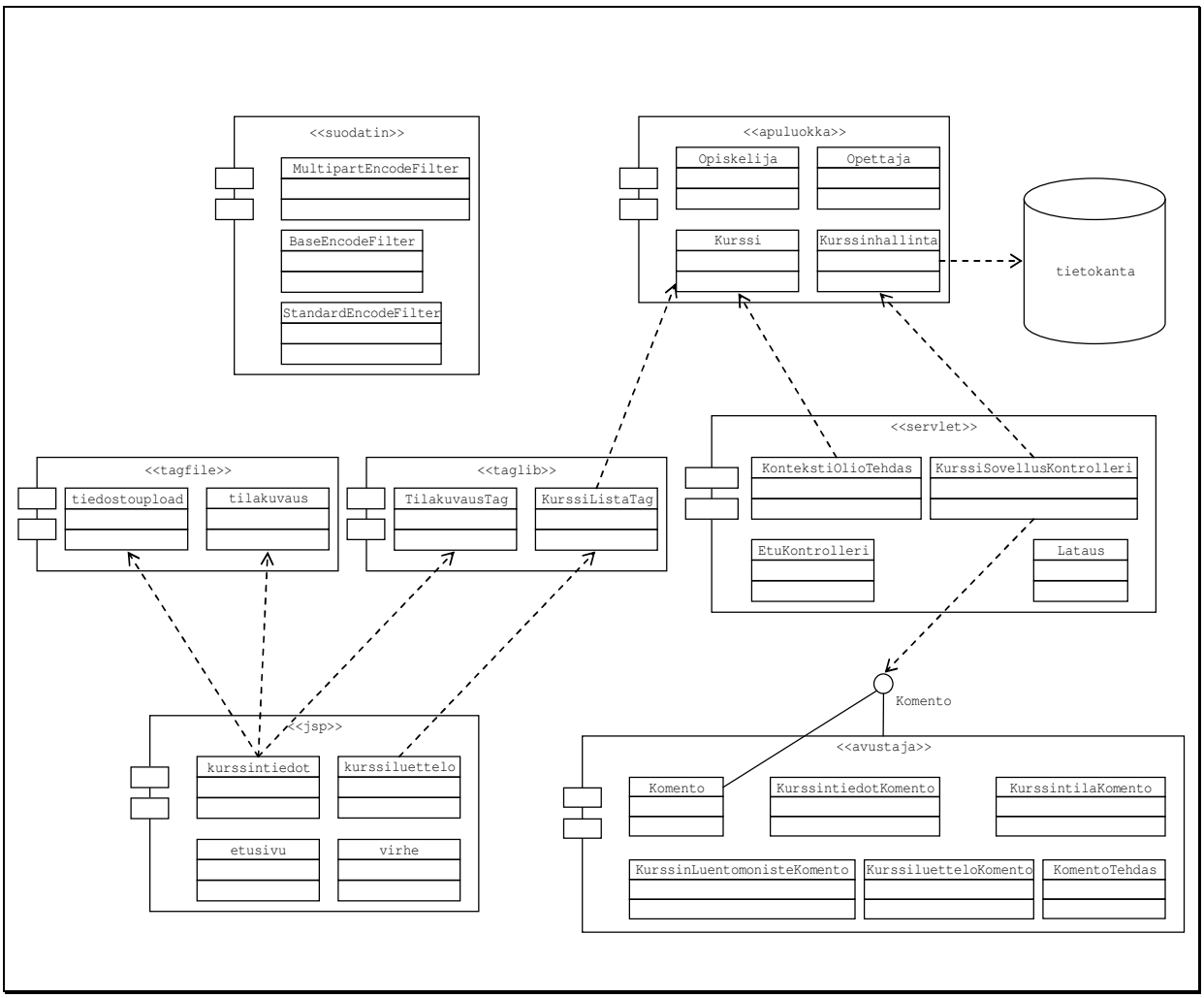
Nock C.: *Data Access Patterns, Database Interactions in object-oriented Applications*. Addison-Wesley, 2004.

Sauter P., Vögler G., Specht G., Flor T.: Extending the MVC design pattern towards a task-oriented development approach for pervasive computing applications. *Interantional conference on architecture of computing systems*, Spinger-Verlag, 2004, 309-321.

Silva E., Moreira D.: WebMODE: A Framework for Development of Web-based Tools for Management of Educational Activities, *Fifth IEEE International Conference on Advanced Learning Technologies (ICALT'05)*, IEEE Computer Society, 922-924, 2005.

Toivonen A. (suom.), Gamma E., Helm R., Johnson R., Vlissides J.: *Olio-ohjelmointi – Suunnittelumallit*. Oy Edita Ab, 2001.

LIITE 1: Kurssinhallintasovelluksen rakenne



Http-palvelupyyntöä vastaanottaessa sovelluspalvelin ohjaa pyynnön suodatinketjulle, mikäli se on konfiguroitu käynnistymään palvelupyynnön kohteena olevassa osoitteessa. Suodattimien konfigurointi on tehty sovelluskohtaiseen web.xml-tiedostoon. Mahdollisten suodattimien jälkeen pyyntö käsitellään EtuKontrollerissa, joka ohjaa pyynnön joko staattiselle html-sivulle tai KurssiSovellusKontrollerille. KurssiSovellusKontrolleri etsii KomentoTehtaan avulla sopivan Komennon, joka suorittaa sille kuuluvan operaation, esim. hakee kurssit tietokannasta apuluokkien avulla, ja palauttaa seuraavan näkymän osoitteen, johon suoritus ohjataan. Näkymiä ovat kurssintiedot.jsp, kurssiluettelo.jsp ja etusivu.html. Jsp-sivut käyttävät tagi-tiedostoja ja tagi-kirjastoja apuna näytön muodostamisessa.

LIITE 2: Kurssinhallintasovellus. Sieppaussuodatin.

```
/**
Sieppaussuodatin
Standard Filter Strategy Alur & al. (2003)
*/

package kurssinhallinta.suodatin;

import javax.servlet.*;

public class BaseEncodeFilter implements Filter {
  private FilterConfig myFilterConfig;

  public void init(FilterConfig filterConfig) {
    this.myFilterConfig = filterConfig;
  }

  public BaseEncodeFilter()     {  }

  public void doFilter( ServletRequest servletRequest, ServletResponse servletResponse,
    FilterChain filterChain) throws java.io.IOException, ServletException {
    filterChain.doFilter(servletRequest, servletResponse);
  }

  public FilterConfig getFilterConfig() {
    return myFilterConfig;
  }

  public void setFilterConfig(
    FilterConfig filterConfig) {
      myFilterConfig = filterConfig;
  }

  public void destroy() {
    this.myFilterConfig = null;
  }
}

/**
Sieppaussuodatin
Standard Filter Strategy Alur & al. (2003)
*/

package kurssinhallinta.suodatin;

import javax.servlet.*;
import javax.servlet.http.*;
import java.util.*;

public class StandardEncodeFilter extends BaseEncodeFilter {

  public StandardEncodeFilter()   {  }

  public void doFilter(ServletRequest servletRequest,
  ServletResponse servletResponse, FilterChain filterChain)
  throws java.io.IOException, ServletException {

    ServletContext servletContext =
      ((HttpServletRequest)servletRequest).getSession().getServletContext();

    String contentType = servletRequest.getContentType();
    if ((contentType == null) ||
      contentType.equalsIgnoreCase("application/x-www-form-urlencoded")) {
      translateParamsToAttributes(servletRequest, servletResponse);
    }

    filterChain.doFilter(servletRequest, servletResponse);
    //servletContext.log("StandardEncodeFilter doFilter done");
  }
```

```
  private void translateParamsToAttributes(
    ServletRequest request, ServletResponse response)
  {
    Enumeration paramNames = request.getParameterNames();

    while (paramNames.hasMoreElements()) {
      String paramName = (String)
      paramNames.nextElement();

      String [] values;
      values = request.getParameterValues(paramName);
      if (values.length == 1)
        request.setAttribute(paramName, values[0]);
      else
        request.setAttribute(paramName, values);
    }
 }
}

/**
Sieppaussuodatin
Standard Filter Strategy Alur & al. (2003)
*/

package kurssinhallinta.suodatin;

import javax.servlet.*;

import javax.servlet.http.*;
import org.apache.commons.httpclient.methods.*;
import org.apache.commons.logging.*;
import org.apache.commons.fileupload.*;
import org.apache.commons.fileupload.servlet.*;
import org.apache.commons.fileupload.disk.*;
import java.text.*;
import java.util.*;
import java.io.*;

public class MultipartEncodeFilter extends BaseEncodeFilter {

  String tempDir = "c:/tmp/";
  String uploadFolder = "c:/tmp/";
  int maxMemorySize = 1 * 1024 * 1024;

  public MultipartEncodeFilter() { }

  public void doFilter(ServletRequest servletRequest,
  ServletResponse servletResponse,FilterChain filterChain)
  throws java.io.IOException, ServletException {

    ServletContext servletContext =
      ((HttpServletRequest)servletRequest).getSession().getServletContext();

    RequestContext requestContext = (RequestContext) new ServletRequestCon-
text((HttpServletRequest)servletRequest);
    FileItemFactory factory = new DiskFileItemFactory(maxMemorySize, new File(tempDir));
    ServletFileUpload upload = new ServletFileUpload(factory);

    boolean isMultipart = upload.isMultipartContent(requestContext);

    if (isMultipart) {
      try {
        upload.setSizeMax(maxMemorySize);

        // Parsitaan palvelupyyntö
        List items = upload.parseRequest((HttpServletRequest) servletRequest);

        // Prosessoidaan ylösladatut kohteet
        Iterator iter = items.iterator();
        while (iter.hasNext()) {
```

```
          FileItem item = (FileItem) iter.next();
          if (item.isFormField()) {
            // Prosessoidaan tavalliset lomakekentät
            String name = item.getFieldName();
            String value = item.getString();
            servletRequest.setAttribute(name, value);
          } else {
            //Prosessoidaan tiedostot
            String fieldName = item.getFieldName();
            String fileNamePath = item.getName();
            String fileName = parseFilename(fileNamePath);
            long sizeInBytes = item.getSize();
            File uploadedFile = new File(uploadFolder + fileName);
            item.write(uploadedFile);
            servletRequest.setAttribute(fieldName, fileName);
          }
   I   }
    } catch (IOException e) {
      servletContext.log("MultipartEncodeFilter, I/O poikkeus: ", e);
    } catch (FileUploadException e) {
      servletContext.log("MultipartEncodeFilter, virhe tiedostoa lukiessa: ", e);
    } catch (java.lang.Exception e) {
      servletContext.log("MultipartEncodeFilter, jokin muu virhe: ", e);
    }
  }
  filterChain.doFilter(servletRequest, servletResponse);
}

public String parseFilename(String filename) {
  String newName = "";
  if (filename.indexOf("\\")!=-1)
    newName = filename.substring(filename.lastIndexOf("\\")+1);
  else
    newName = filename;
    return newName;
  }
}
```

LIITE 3: Kurssinhallintasovellus. Näkymäavustaja.

```
<!-- etusivu.jsp -->
<html>
<head><title>Kurssinhallinta</title></head>
<body>
<h1>Kurssinhallinta</h1>
<table border="0" width="100%" cellspacing="2" cellpadding="3">
   <tr>
      <td><a href="kontrolleri?toiminto=kurssiluettelo">Listaa kurssit</a></td>
   </tr>
   <tr>
      <td>Hae kurssi</td>
   </tr>
</table>
</body>
</html>

<!-- kurssiluettelo.jsp
Näkymäavustaja, Custom Tag Helper Strategy.
Tagikirjaston kuvaus on tiedostossa /WEB-INF/lib/kurssinhallinta.tld
Kustomoidut tagit on paketissa kurssinhallinta.taglib
-->

<%@ page import="kurssinhallinta.apuluokka.Kurssi,java.util.*" %>
<%@ taglib prefix="kh" uri="/WEB-INF/lib/kurssinhallinta.tld" %>

<html>
<head><title>Kurssinhallinta / Kurssiluettelo</title></head>
<body>
<p><a href="kontrolleri?toiminto=etusivu">Takaisin etusivulle</a></p>
<%
   List kurssit = (List) request.getAttribute("kurssit");
%>

<kh:kurssilista kurssit="<%=kurssit%>"/>

</body>
</html>

<!-- kurssintiedot.jsp
Näkymäavustaja, Tag File Helper Strategy.
Tagikirjastot on hakemistossa /WEB-INF/tags/
Tagi otetaan käyttöön viittaamalla tagikirjaston tiedoston nimeen,
joka viittaa tilakuvaus.tag-tiedostoon.
Tagille välitetään arvo attribuuttina, jonka perusteella
se suorittaa muotoilun näytölle.
-->
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>
<%@ taglib prefix="tk" tagdir="/WEB-INF/tags/" %>

<html>
<head><title>Kurssinhallinta / Kurssin tiedot</title></head>
<body>
<p><a href="kontrolleri?toiminto=etusivu">Takaisin etusivulle</a>
<a href="kontrolleri?toiminto=kurssiluettelo">Takaisin kurssilistaukseen</a></p>

<h1>Kurssin tiedot</h1>
<h3>Kurssi</h3>

Nimi: <c:out value="${kurssi.nimi}"/></br>
Aloituspvm: <c:out value="${kurssi.aloitusPvm}"/></br>
Opintoviikot: <c:out value="${kurssi.opintopisteet}"/></br>
Opettaja: <c:out value="${opettaja.etunimi}"/>, <c:out value="${opettaja.sukunimi}"/>

<!-- Esimerkin vuoksi käytetään lomakkeen muodostamiseen tagia -->
<tk:tiedostoupload>
  <jsp:attribute name="osoite">
     <c:out value="kontrolleri"/>
  </jsp:attribute>
```

```
  <jsp:attribute name="otsikko">
     <c:out value="Luentomoniste:"/>
  </jsp:attribute>
  <jsp:attribute name="tiedosto">
     <c:out value="${kurssi.luentomoniste}"/>
  </jsp:attribute>
  <jsp:attribute name="toiminto">
     <c:out value="paivitakurssinluentomoniste"/>
  </jsp:attribute>
  <jsp:attribute name="idnimi">
     <c:out value="kurssiId"/>
  </jsp:attribute>
  <jsp:attribute name="idarvo">
     <c:out value="${kurssi.kurssiId}"/>
  </jsp:attribute>
</tk:tiedostoupload>
</br>

Tila:
<!-- Käytetään tilakuvaus-tagia kurssin tilan näyttämiseen -->
<tk:tilakuvaus>
  <jsp:attribute name="tilakuvausStr">
     <c:out value="${kurssi.tilakuvaus}"/>
  </jsp:attribute>
</tk:tilakuvaus>


<!-- Normaali lomakkeen muodostus -->
<form action="kontrolleri" method="post">
  <input type="hidden" name="toiminto" value="muutakurssintilaa">
  <input type="hidden" name="kurssiId" value="<c:out value="${kurssi.kurssiId}"/>">
  <input type="submit" name="aloitettu" value="Muuta aloitetuksi"/>
   
  <input type="submit" name="paattynyt" value="Muuta päättyneeksi"/>
</form>

</br>

<h3>Opiskelijat</h3>
<table border="1" cellspacing="2" cellpadding="3">
   <tr>
      <th>Etunimi</th>
      <th>Sukunimi</th>
      <th>Opiskelijanumero</th>
   </tr>
            <!-- JSTL silmukkarakenne -->
   <c:forEach items="${opiskelijat}" var="opiskelija">
     <tr>
       <td><c:out value="${opiskelija.etunimi}"/></td>
       <td><c:out value="${opiskelija.sukunimi}"/></td>
       <td><c:out value="${opiskelija.opiskelijaNro}"/></td>
            </tr>
   </c:forEach>
</table>
</body>
</html>

<!-- virhe.jsp -->
<html>
<head><title>Kurssinhallinta / Virhe</title></head>
<body>
<h1>Virhe</h1>
<p>Sovelluksessa tapahtui virhe. Tekniset tiedot: </p>
<p>
<pre>
<%
out.flush();
((Exception) request.getAttribute("virhe")).printStackTrace(response.getWriter());
%>
</pre>
</p>
```

```
</body>
</html>

tiedostoupload.tag:
<%@ attribute name="osoite" required="true" %>
<%@ attribute name="otsikko" required="true" %>
<%@ attribute name="tiedosto" required="true" %>
<%@ attribute name="toiminto" required="true" %>
<%@ attribute name="idnimi" required="true" %>
<%@ attribute name="idarvo" required="true" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>


<form method="post" action="<c:out value="${osoite}"/>" enctype="multipart/form-data">
<input type="hidden" name="toiminto" value="<c:out value="${toiminto}"/>">
<input type="hidden" name="<c:out value="${idnimi}"/>" value="<c:out
value="${idarvo}"/>">
<c:if test="${tiedosto == ''}">
  <c:out value="${otsikko}"/> <input type="file" name="tiedosto">
  <input type="submit" name="tiedosto_lisaa" value="Lähetä">
</c:if>
<c:if test="${tiedosto != ''}">
  <c:out value="${otsikko}"/>
  <a href="<c:out value="${osoite}"/>?toiminto=lataus&tiedosto=
  <c:out value="${tiedosto}"/>" target='_blank'>
  <c:out value="${tiedosto}"/></a>
  <input type="submit" name="tiedosto_poista" value="Poista tiedosto">
</c:if>
</form>

Tilakuvaus.tag:
<%@ attribute name="tilakuvausStr" required="true" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core" %>

<font color="gray">
<c:if test="${tilakuvausStr == 'aloitettu'}">
   </font><font color="green">
</c:if>
<c:if test="${tilakuvausStr == 'päättynyt'}">
   </font><font color="red">
</c:if>
<c:out value="${tilakuvausStr}"/>
</font>

Kurssinhallinta.tld
<?xml version="1.0" encoding="UTF-8"?>
<taglib xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
http://java.sun.com/xml/ns/j2ee/web-jsptaglibrary_2_0.xsd"
    version="2.0">
  <tlib-version>2.0</tlib-version>
  <short-name>Kurssinhallinta</short-name>
  <tag>
    <name>tilakuvaus</name>
    <tag-class>kurssinhallinta.taglib.TilakuvausTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <name>tila</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
    </attribute>
  </tag>
  <tag>
    <name>kurssilista</name>
    <tag-class>kurssinhallinta.taglib.KurssiListaTag</tag-class>
    <body-content>empty</body-content>
    <attribute>
      <name>kurssit</name>
      <required>true</required>
      <rtexprvalue>true</rtexprvalue>
```

```
    </attribute>
  </tag>
</taglib>

/**
Näkymäavustaja, Custom Tag Helper Strategy.
Tagikirjaston kuvaus on tiedostossa /WEB-INF/lib/kurssinhallinta.tld
Kustomoitu tagi-luokka KurssiListaTag
*/

package kurssinhallinta.taglib;

import java.io.IOException;
import javax.servlet.jsp.JspContext;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.SimpleTagSupport;

public class TilakuvausTag extends SimpleTagSupport {
  private String tila;

  // Varsinainen suoritusmetodi, joka hoitaa tulostuksen
  public void doTag() {
    JspContext jspCtx = getJspContext();
    try {
      JspWriter out = jspCtx.getOut();
      if (tila.equals("aloittamatta")) {
        out.println("<font color=\"blue\">aloittamatta</font>");
      } else {
        out.println(tila);
      }
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

  // Tag-elementin attribuutin arvo ohjautuu tämän metodin kutsuksi
  public void setTila(String tila) {
    this.tila = tila;
  }
}

/**
Näkymäavustaja, Custom Tag Helper Strategy.
Tagikirjaston kuvaus on tiedostossa /WEB-INF/lib/kurssinhallinta.tld
Kustomoitu tagi-luokka KurssiListaTag
*/

package kurssinhallinta.taglib;

import java.io.IOException;
import java.util.List;
import java.util.Iterator;
import javax.servlet.jsp.JspContext;
import javax.servlet.jsp.JspWriter;
import javax.servlet.jsp.tagext.SimpleTagSupport;
import kurssinhallinta.apuluokka.*;

public class KurssiListaTag extends SimpleTagSupport {
  private List kurssit;

  // Varsinainen suoritusmetodi, joka hoitaa tulostuksen
  public void doTag() {
    JspContext jspCtx = getJspContext();
    try {
      JspWriter out = jspCtx.getOut();
      out.println("<h1>Kurssiluettelo</h1>");
      out.println("<table border=\"1\" cellspacing=\"2\" cellpadding=\"3\">");
      for (Iterator kurssiIter = kurssit.iterator(); kurssiIter.hasNext();) {
        Kurssi kurssi = (Kurssi) kurssiIter.next();
        out.println("<tr>");
        out.println("<td><a href=\"kontrolleri?toiminto=kurssitiedot&kurssiId=" +
```

```
        kurssi.getKurssiId() + "\">" + kurssi.getNimi() +"</a></td>");
        out.println("<td>" + kurssi.getAloitusPvm() + "</td>");
        if (kurssi.getTilakuvaus().equals("aloittamatta")) {
          out.println("<td><font color=\"blue\">aloittamatta</font></td>");
        } else {
          out.println("<td>" + kurssi.getTilakuvaus() + "</td>");
        }
        out.println("<tr>");
      }
      out.println("</table>");
    } catch (IOException ex) {
      ex.printStackTrace();
    }
  }

  // Tag-elementin attribuutin arvo ohjautuu tämän metodin kutsuksi
  public void setKurssit(List kurssit) {
    this.kurssit = kurssit;
  }
}
```

LIITE 4: Kurssinhallintasovellus. Kontekstiolio.

```
/**
Kontekstiolio-suunnittelumalli.
Tehdas, jonka avulla luodaan kontekstiolioita
*/

package kurssinhallinta.servlet;

import javax.servlet.http.HttpServletRequest;
import kurssinhallinta.apuluokka.*;

public class KontekstiOlioTehdas {

  // Konstruktori on private, jolloin sitä ei voi turhaan instantioida
  private KontekstiOlioTehdas() { }
  public static Kurssi haeKurssi(HttpServletRequest req) throws AlustusException {
    //Haetaan kurssin id
    int kurssiId = Integer.parseInt((String) req.getAttribute("kurssiId"));
    String moniste = (String) req.getAttribute("tiedosto");

    // Selvitetään kurssin tila
    int uusiTila = 1;
    if ((String) req.getAttribute("paattynyt")!= null) {
      uusiTila = 2;
    }
    Kurssi kurssi = new Kurssi(kurssiId, "", "", uusiTila, 0.0f, moniste);
    return kurssi;
  }
}

/**
Kontekstiolio-suunnittelumalli.
*/
package kurssinhallinta.apuluokka;

public class Kurssi {
  private int kurssiId;
  private String nimi;
  private String aloitusPvm;
  private int tila;
  private float opintopisteet;
  private String luentomoniste;

  public Kurssi(int kurssiId, String nimi, String aloitusPvm, int tila,
      float opintopisteet, String luentomoniste) {
    this.kurssiId = kurssiId;
    this.nimi = nimi;
    this.aloitusPvm = aloitusPvm;
    this.tila = tila;
    this.opintopisteet = opintopisteet;
    this.luentomoniste = luentomoniste;
  }

  public String getNimi() {
    return nimi;
  }

  public void setNimi(String nimi) {
    this.nimi = nimi;
  }

  public float getOpintopisteet() {
    return opintopisteet;
  }

  public void setOpintopisteet(float opintopisteet) {
    this.opintopisteet = opintopisteet;
  }
```

```
  public String getTilakuvaus() {
    String tilaStr = null;
    switch (tila) {
    case 0:
      tilaStr = "aloittamatta";
      break;
    case 1:
      tilaStr = "aloitettu";
      break;
    case 2:
      tilaStr = "päättynyt";
      break;
    default:
      tilaStr = "aloittamatta";
    }
    return tilaStr;
  }

  public int getTila() {
    return tila;
  }

  public void setTila(int tila) {
    this.tila = tila;
  }

  public String getAloitusPvm() {
    return aloitusPvm;
  }

  public void setAloitusPvm(String aloitusPvm) {
    this.aloitusPvm = aloitusPvm;
  }

  public String getLuentomoniste() {
    return this.luentomoniste;
  }

  public void setLuentomoniste(String luentomoniste) {
    this.luentomoniste = luentomoniste;
  }

  public int getKurssiId() {
    return kurssiId;
  }

  public void setKurssiId(int kurssiId) {
    this.kurssiId = kurssiId;
  }
}

/**
Kontekstiolio-suunnittelumalli.
*/
package kurssinhallinta.apuluokka;

public class Opettaja {
  private int opettajaId;
  private String etunimi;
  private String sukunimi;

  public Opettaja(int opettajaId, String etunimi, String sukunimi) {
    this.opettajaId = opettajaId;
    this.etunimi = etunimi;
    this.sukunimi = sukunimi;
  }

  public int getId() {
    return opettajaId;
  }
```

```
  public void setId(int opettajaId) {
    this.opettajaId = opettajaId;
    }

  public String getEtunimi() {
    return etunimi;
  }

  public void setEtunimi(String etunimi) {
    this.etunimi = etunimi;
  }

  public String getSukunimi() {
    return sukunimi;
  }

  public void setSukunimi(String sukunimi) {
    this.sukunimi = sukunimi;
  }
}

/**
Kontekstiolio-suunnittelumalli.
*/
package kurssinhallinta.apuluokka;

public class Opiskelija {
  private int opiskelijaId;
  private String etunimi;
  private String sukunimi;
  private String opiskelijaNro;

  public Opiskelija(int opiskelijaId, String etunimi, String sukunimi, String
      opiskelijaNro) {
    this.opiskelijaId = opiskelijaId;
    this.etunimi = etunimi;
    this.sukunimi = sukunimi;
    this.opiskelijaNro = opiskelijaNro;
  }

  public int getId() {
    return opiskelijaId;
  }

  public void setId(int opiskelijaId) {
    this.opiskelijaId = opiskelijaId;
  }

  public String getEtunimi() {
    return etunimi;
  }

  public void setEtunimi(String etunimi) {
    this.etunimi = etunimi;
  }

  public String getSukunimi() {
    return sukunimi;
  }

  public void setSukunimi(String sukunimi) {
    this.sukunimi = sukunimi;
  }

  public String getOpiskelijaNro() {
    return opiskelijaNro;
  }

  public void setOpiskelijaNro(String opiskelijaNro) {
    this.opiskelijaNro = opiskelijaNro;
  }}
```

LIITE 5: Kurssinhallintasovellus. Sovelluskontrolleri.

```
/**
Sovelluskontrolleri
Hallinnoi toimintoja ja näkymiä
*/

package kurssinhallinta.servlet;

import java.io.IOException;
import java.util.List;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import kurssinhallinta.apuluokka.*;
import kurssinhallinta.avustaja.*;

public class KurssiSovellusKontrolleri {

  private ServletContext servletCtx;

  public KurssiSovellusKontrolleri(ServletContext servletCtx) {
    this.servletCtx = servletCtx;
  }

  public void palvelePyynto(HttpServletRequest req, HttpServletResponse res, String
      toiminto) throws IOException, ServletException {
    try {

      //Haetaan suoritettava toiminto komentotehtaalta
      KomentoTehdas komentoTehdas = KomentoTehdas.haeInstanssi();
      Komento komento = komentoTehdas.haeKomento(toiminto);

      //Suoritetaan toiminto, joka palauttaa näytön osoitteen
      String view = komento.suorita(req);

      //Välitetään komennon palauttamalle näytölle
        req.getRequestDispatcher(view).forward(req, res);
    } catch (HakuException ex) {
      naytaVirhesivu(req,res,new Exception("Kurssilistaa ei ole!"));
    }
  }

  public void naytaVirhesivu(HttpServletRequest req, HttpServletResponse res,
      Exception ex) throws IOException, ServletException {
    this.servletCtx.log("Hakuvirhe", ex);
    ex.printStackTrace();
    req.setAttribute("virhe", ex);
    req.getRequestDispatcher("/virhe.jsp").forward(req, res);
  }
}


/**
Komento-suunnittelumalli.
Rajapinta, jonka suorita-metodin konkreettinen Komento-luokka toteuttaa
*/

package kurssinhallinta.avustaja;

import javax.servlet.http.HttpServletRequest;
import kurssinhallinta.apuluokka.*;

public interface Komento {
   public String suorita(HttpServletRequest req) throws HakuException;
}

/**
Yksinkertaistettu toteutus
```

```
Abstrakti tehdas-suunnittelumallista käyttäen Tehdasmetodi-suunnittelumallia
*/

package kurssinhallinta.avustaja;

import java.util.HashMap;

public class KomentoTehdas {

  HashMap komentoVarasto = null;
  private static KomentoTehdas instanssi = null;

  protected KomentoTehdas() {
    // Luodaan komentoVaraston sisältö tässä yksinkertaistamisen vuoksi
    komentoVarasto = new HashMap();
    komentoVarasto.put("kurssiluettelo",new KurssiluetteloKomento());
    komentoVarasto.put("kurssitiedot",new KurssintiedotKomento());
    komentoVarasto.put("muutakurssintilaa",new KurssintilaKomento());
    komentoVarasto.put("paivitakurssinluentomoniste",new KurssinluentomonisteKomento());
  }

  public static KomentoTehdas haeInstanssi() {
    if (instanssi == null) instanssi = new KomentoTehdas();
      return instanssi;
    }

    public Komento haeKomento(String komentoNimi) {
      if ((komentoVarasto != null) && komentoVarasto.containsKey(komentoNimi)) {
        return (Komento)komentoVarasto.get(komentoNimi);
      } else {
        return null;
    }
  }
}

/**
Komento-suunnittelumalli.
Konkreettinen luokka, joka toteuttaa Komento-rajapinnan suorita-metodin
*/

package kurssinhallinta.avustaja;

import javax.servlet.http.HttpServletRequest;
import java.util.List;
import kurssinhallinta.apuluokka.*;

public class KurssiluetteloKomento implements Komento {
  public KurssiluetteloKomento() { }

  public String suorita(HttpServletRequest req) throws HakuException {
    //Näytön osoite, joka palautetaan onnistuneen suorituksen päätteeksi
    String naytto = "/kurssiluettelo.jsp";
    try {
      Kurssihallinta kurssihallinta = new Kurssihallinta();
      List kurssit = kurssihallinta.haeKurssit();
      req.setAttribute("kurssit", kurssit);
    }catch (Exception ex) {
      throw new HakuException(ex);
    }
    return naytto;
  }
}

/**
Komento-suunnittelumalli.
Konkreettinen luokka, joka toteuttaa Komento-rajapinnan suorita-metodin
*/

package kurssinhallinta.avustaja;

import javax.servlet.http.HttpServletRequest;
```

```
import java.util.List;
import kurssinhallinta.servlet.KontekstiOlioTehdas;
import kurssinhallinta.apuluokka.*;

public class KurssinluentomonisteKomento implements Komento {
  public KurssinluentomonisteKomento() { }

  public String suorita(HttpServletRequest req) throws HakuException {
    //Näytön osoite, joka palautetaan onnistuneen suorituksen päätteeksi
    String naytto = "/kurssintiedot.jsp";
    try {
      Kurssihallinta kurssihallinta = new Kurssihallinta();
      int kurssiId = Integer.parseInt((String) req.getAttribute("kurssiId"));
      // Uuden Kurssi-olion luontiin käytetään KontekstiOlioTehdas-luokkaa,
      // jossa luetaan palvelupyynnön parametrit
      kurssihallinta.paivitaKurssinLuentomoniste(
        KontekstiOlioTehdas.haeKurssi(req));
      Kurssi kurssi = kurssihallinta.haeKurssi(kurssiId);
      Opettaja opettaja = kurssihallinta.haeKurssinOpettaja(kurssiId);
      List opiskelijat = kurssihallinta.haeKurssinOpiskelijat(kurssiId);
      req.setAttribute("kurssi", kurssi);
      req.setAttribute("opettaja", opettaja);
      req.setAttribute("opiskelijat", opiskelijat);
    }catch (Exception ex) {
      throw new HakuException(ex);
    }
    return naytto;
  }
}

/**
Komento-suunnittelumalli.
Konkreettinen luokka, joka toteuttaa Komento-rajapinnan suorita-metodin
*/

package kurssinhallinta.avustaja;

import javax.servlet.http.HttpServletRequest;
import java.util.List;
import kurssinhallinta.apuluokka.*;

public class KurssintiedotKomento implements Komento {
  public KurssintiedotKomento() { }

  public String suorita(HttpServletRequest req) throws HakuException {
    //Näytön osoite, joka palautetaan onnistuneen suorituksen päätteeksi
    String naytto = "/kurssintiedot.jsp";
    try {
      Kurssihallinta kurssihallinta = new Kurssihallinta();
      int kurssiId = Integer.parseInt((String) req.getAttribute("kurssiId"));
      Kurssi kurssi = kurssihallinta.haeKurssi(kurssiId);
      Opettaja opettaja = kurssihallinta.haeKurssinOpettaja(kurssiId);
      List opiskelijat = kurssihallinta.haeKurssinOpiskelijat(kurssiId);
      req.setAttribute("kurssi", kurssi);
      req.setAttribute("opettaja", opettaja);
      req.setAttribute("opiskelijat", opiskelijat);
    }catch (Exception ex) {
      throw new HakuException(ex);
    }
    return naytto;
  }
}

/**
Komento-suunnittelumalli.
Konkreettinen luokka, joka toteuttaa Komento-rajapinnan suorita-metodin
*/

package kurssinhallinta.avustaja;

import javax.servlet.http.HttpServletRequest;
```

```
import java.util.List;
import kurssinhallinta.servlet.KontekstiOlioTehdas;
import kurssinhallinta.apuluokka.*;

public class KurssintilaKomento implements Komento {
  public KurssintilaKomento() { }

  public String suorita(HttpServletRequest req) throws HakuException {
    //Näytön osoite, joka palautetaan onnistuneen suorituksen päätteeksi
    String naytto = "/kurssintiedot.jsp";
    try {
      Kurssihallinta kurssihallinta = new Kurssihallinta();
      int kurssiId = Integer.parseInt((String) req.getAttribute("kurssiId"));
      // Uuden Kurssi-olion luontiin käytetään KontekstiOlioTehdas-luokkaa,
      // jossa luetaan palvelupyynnön parametrit
      kurssihallinta.muutaKurssinTilaa(
        KontekstiOlioTehdas.haeKurssi(req));
      Kurssi kurssi = kurssihallinta.haeKurssi(kurssiId);
      Opettaja opettaja = kurssihallinta.haeKurssinOpettaja(kurssiId);
      List opiskelijat = kurssihallinta.haeKurssinOpiskelijat(kurssiId);
      req.setAttribute("kurssi", kurssi);
      req.setAttribute("opettaja", opettaja);
      req.setAttribute("opiskelijat", opiskelijat);
    }catch (Exception ex) {
      throw new HakuException(ex);
    }
    return naytto;
  }
}
```

LIITE 6: Kurssinhallintasovellus. Etukontrolleri.

```
/**
Etukontrolleri-suunnittelumalli.
Käytetään Välittäjänäkymä-suunnittelumallin dispatcher in controller strategiaa
ja Palvelu työstäjälle-suunnittelumallia.
Palvelu työstäjälle-mallissa Etukontrolleri ottaa vastaan palvelupyynnöt ja
delegoi kontrollin KurssiSovellusKontrollerille (Sovelluskontrolleri),
joka hakee avustajien avulla tarvittavat tiedot ja välittää näkymälle
*/

package kurssinhallinta.servlet;

import java.io.IOException;
import java.util.List;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;


public class EtuKontrolleri extends HttpServlet {
  private KurssiSovellusKontrolleri kurssiSK;

  public void init() {
    try {
      kurssiSK = new KurssiSovellusKontrolleri(this.getServletContext());
    } catch (Exception ex) {
      this.log("Alustusvirhe", ex);
      ex.printStackTrace();
    }
  }

  public void doGet(HttpServletRequest req, HttpServletResponse res)
      throws IOException, ServletException {
    String toiminto = (String) req.getAttribute("toiminto");
    // Annetaan toiminnolle oletusarvo tarvittaessa
    toiminto = (toiminto != null ? toiminto : "etusivu");

    if (toiminto.equals("etusivu")) {
      req.getRequestDispatcher("/etusivu.jsp").forward(req, res);
    } else if (toiminto.equals("lataus")) {
      try {
        req.setAttribute("polku","c:\\tmp");
        req.getRequestDispatcher("Lataus").forward(req, res);
      } catch (Exception ex) {
        kurssiSK.naytaVirhesivu(req, res, ex);
      }
    } else {
      kurssiSK.palvelePyynto(req, res, toiminto);
    }
  }

  public void doPost(HttpServletRequest req, HttpServletResponse res)
      throws IOException, ServletException {
    doGet(req, res);
  }
}
```

LIITE 7: Kurssinhallintasovellus. Apuluokat

```
package kurssinhallinta.apuluokka;

import javax.naming.InitialContext;
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
import javax.sql.DataSource;

public class Kurssihallinta {
  private DataSource tietolahde;

  public Kurssihallinta() throws AlustusException {
    try {
      otaYhteys();
    } catch (Exception ex) {
       throw new AlustusException(ex);
    }
  }

  public void otaYhteys() throws Exception {
    if (tietolahde==null) {
      javax.naming.Context ctx = new javax.naming.InitialContext();
      String kanta = (String) ctx.lookup("java:/comp/env/kanta");
      tietolahde = (javax.sql.DataSource) ctx.lookup("java:/comp/env/"+kanta);
    }
  }

  public List<Opiskelija> haeOpiskelijat() throws HakuException {
    List<Opiskelija> opiskelijat = new ArrayList<Opiskelija>();
    Connection yhteys = null;
    Statement lause = null;
    ResultSet tulosjoukko = null;
    try {
      yhteys = tietolahde.getConnection();
      lause = yhteys.createStatement();
      tulosjoukko = lause.executeQuery(
        "SELECT sukunimi, etunimi, opiskelijanro, id" +
        " FROM opiskelija ORDER BY sukunimi, etunimi;");
      while (tulosjoukko.next()) {
        opiskelijat.add(new Opiskelija(tulosjoukko.getInt("id"),
        tulosjoukko.getString("etunimi"),
        tulosjoukko.getString("sukunimi"),
        tulosjoukko.getString("opiskelijanro")));
      }
    } catch (Exception ex) {
      throw new HakuException(ex);
    } finally {
      suljeYhteys(yhteys,lause,tulosjoukko);
    }
    return opiskelijat;
  }

  public List<Opiskelija> haeKurssinOpiskelijat(int id) throws HakuException {
    List<Opiskelija> opiskelijat = new ArrayList<Opiskelija>();
    Connection yhteys = null;
    PreparedStatement lause = null;
    ResultSet tulosjoukko = null;
    try {
      yhteys = tietolahde.getConnection();
      lause = yhteys.prepareStatement(
      "SELECT o.sukunimi, o.etunimi, o.opiskelijanro, o.id" +
      " FROM opiskelija o, kurssinopiskelijat k WHERE k.opiskelija = o.id" +
      " AND k.kurssi = ? ORDER BY o.sukunimi, o.etunimi;");
      lause.setInt(1, id);
      tulosjoukko = lause.executeQuery();
```

```
    while (tulosjoukko.next()) {
      opiskelijat.add(new Opiskelija(tulosjoukko.getInt("o.id"),
       tulosjoukko.getString("o.etunimi"),
       tulosjoukko.getString("o.sukunimi"),
       tulosjoukko.getString("o.opiskelijanro")));
    }
  } catch (Exception ex) {
    throw new HakuException(ex);
  } finally {
    suljeYhteys(yhteys,lause,tulosjoukko);
  }
  return opiskelijat;
}

public List<Kurssi> haeKurssit() throws HakuException {
  List<Kurssi> kurssit = new ArrayList<Kurssi>();
  Connection yhteys = null;
  Statement lause = null;
  ResultSet tulosjoukko = null;
  try {
    yhteys = tietolahde.getConnection();
    lause = yhteys.createStatement();
    tulosjoukko = lause.executeQuery(
      "SELECT nimi, aloituspvm, tila, opintopisteet, luentomoniste, id" +
       " FROM kurssi ORDER BY nimi, aloituspvm;");
     while (tulosjoukko.next()) {
       kurssit.add(new Kurssi(tulosjoukko.getInt("id"),
       tulosjoukko.getString("nimi"),
       tulosjoukko.getString("aloituspvm"),
       tulosjoukko.getInt("tila"),
       tulosjoukko.getFloat("opintopisteet"),
       tulosjoukko.getString("luentomoniste")));
     }
  } catch (Exception ex) {
    throw new HakuException(ex);
  } finally {
    suljeYhteys(yhteys,lause,tulosjoukko);
  }
  return kurssit;
}

public Opettaja haeOpettaja(int id) throws HakuException {
  Opettaja opettaja = null;
  Connection yhteys = null;
  PreparedStatement lause = null;
  ResultSet tulosjoukko = null;
  try {
    yhteys = tietolahde.getConnection();
    lause = yhteys.prepareStatement(
    "SELECT sukunimi, etunimi, id" +
    " FROM Opettaja WHERE id=?;");
    lause.setInt(1, id);
    tulosjoukko = lause.executeQuery();
    if (tulosjoukko.next()) {
      opettaja = new Opettaja(tulosjoukko.getInt("id"),
      tulosjoukko.getString("etunimi"),
       tulosjoukko.getString("sukunimi"));
    }
  } catch (Exception ex) {
    throw new HakuException(ex);
  } finally {
    suljeYhteys(yhteys,lause,tulosjoukko);
  }
  return opettaja;
}

public Opettaja haeKurssinOpettaja(int id) throws HakuException {
  Opettaja opettaja = null;
  Connection yhteys = null;
  PreparedStatement lause = null;
  ResultSet tulosjoukko = null;
```

```
  try {
    yhteys = tietolahde.getConnection();
    lause = yhteys.prepareStatement(
    "SELECT o.sukunimi, o.etunimi, o.id" +
    " FROM Opettaja o, kurssi k WHERE" +
    " k.opettaja = k.opettaja and k.id=?;");
    lause.setInt(1, id);
    tulosjoukko = lause.executeQuery();
    if (tulosjoukko.next()) {
      opettaja = new Opettaja(tulosjoukko.getInt("id"),
      tulosjoukko.getString("etunimi"),
      tulosjoukko.getString("sukunimi"));
    }
  } catch (Exception ex) {
    throw new HakuException(ex);
  } finally {
    suljeYhteys(yhteys,lause,tulosjoukko);
  }
  return opettaja;
}

public Kurssi haeKurssi(int id) throws HakuException {
  Kurssi kurssi = null;
  Connection yhteys = null;
  PreparedStatement lause = null;
  ResultSet tulosjoukko = null;
  try {
    yhteys = tietolahde.getConnection();
    lause = yhteys.prepareStatement(
    "SELECT nimi, aloituspvm, tila, opintopisteet, opettaja, luentomoniste, id" +
    " FROM Kurssi WHERE id=?;");
    lause.setInt(1, id);
    tulosjoukko = lause.executeQuery();
    if (tulosjoukko.next()) {
      kurssi = new Kurssi(tulosjoukko.getInt("id"),
      tulosjoukko.getString("nimi"),
      tulosjoukko.getString("aloituspvm"),
      tulosjoukko.getInt("tila"),
      tulosjoukko.getFloat("opintopisteet"),
      tulosjoukko.getString("luentomoniste"));
    }
  } catch (Exception ex) {
    throw new HakuException(ex);
  } finally {
    suljeYhteys(yhteys,lause,tulosjoukko);
  }
  return kurssi;
}

public void muutaKurssinTilaa(Kurssi kurssi) throws HakuException {
  Connection yhteys = null;
  PreparedStatement lause = null;
  try {
    yhteys = tietolahde.getConnection();
    lause = yhteys.prepareStatement(
    "UPDATE kurssi SET tila=? WHERE id=?;");
    lause.setInt(1, kurssi.getTila());
    lause.setInt(2, kurssi.getKurssiId());
    lause.execute();
  } catch (Exception ex) {
    throw new HakuException(ex);
  } finally {
    suljeYhteys(yhteys,lause,null);
  }
}

public void paivitaKurssinLuentomoniste(Kurssi kurssi) throws HakuException {
  Connection yhteys = null;
  PreparedStatement lause = null;
  try {
    yhteys = tietolahde.getConnection();
```

```
      lause = yhteys.prepareStatement(
      "UPDATE kurssi SET luentomoniste=? WHERE id=?;");
      lause.setString(1, kurssi.getLuentomoniste());
      lause.setInt(2, kurssi.getKurssiId());
      lause.execute();
    } catch (Exception ex) {
      throw new HakuException(ex);
    } finally {
      suljeYhteys(yhteys,lause,null);
    }
  }

  private void suljeYhteys(Connection yhteys, Statement lause, ResultSet tulosjoukko) {
    if (tulosjoukko != null) {
      try {
        tulosjoukko.close();
      } catch (Exception ex) { ex.printStackTrace(); }
    }
    if (lause != null) {
      try {
        lause.close();
      } catch (Exception ex) { ex.printStackTrace(); }
    }
    if (yhteys != null) {
      try {
        yhteys.close();
      } catch (Exception ex) { ex.printStackTrace(); }
    }
  }

  private void suljeYhteys(Connection yhteys, PreparedStatement lause,
      ResultSet tulosjoukko) {

    if (tulosjoukko != null) {
      try {
        tulosjoukko.close();
      } catch (Exception ex) { ex.printStackTrace(); }
    }
    if (lause != null) {
      try {
        lause.close();
      } catch (Exception ex) { ex.printStackTrace(); }
    }
    if (yhteys != null) {
      try {
        yhteys.close();
      } catch (Exception ex) { ex.printStackTrace(); }
     }
  }

}

package kurssinhallinta.apuluokka;

public class AlustusException extends Exception {

  public AlustusException(String kuvaus) {
    super(kuvaus);
  }

  public AlustusException(Throwable poikkeus) {
    super(poikkeus);
  }
}
package kurssinhallinta.apuluokka;

public class HakuException extends Exception {

  public HakuException(String kuvaus) {
    super(kuvaus);
  }
```

```
  public HakuException(Throwable poikkeus) {
    super(poikkeus);
  }
}

package kurssinhallinta.servlet;

import java.io.*;
import java.util.Hashtable;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.servlet.*;
import javax.servlet.http.*;

public class Lataus extends HttpServlet {
             protected ServletContext context;
  protected Hashtable mimeTypes;

  public Lataus(){ }

  public void init(ServletConfig config) throws ServletException {
    try {
      super.init(config);
      initMimeTypes();
    }
    catch(Exception exception) { }
  }

  public void service(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException {
    try {
      String did = (String) req.getAttribute("tiedosto");
      String path = (String) req.getAttribute("polku");
      String fileName = did;
      String file = path + File.separator + did;

      File ff = new File(file);
      res.setContentType(getMimeType(
        fileName.substring(fileName.lastIndexOf(".") + 1).toLowerCase()));

      res.setHeader("Content-Length", "" + ff.length());
      res.setHeader("Content-Disposition", "filename=\""+ fileName + "\"");
      if(res.containsHeader("Cache-Control")) res.setHeader("Cache-Control", null);
      ServletOutputStream out = res.getOutputStream();
      FileInputStream in = new FileInputStream(ff);
      copy(in, out);
      in.close();
      out.close();
    } catch(Exception e) {
      res.sendError(403, e.toString());
    }
  }

  public static void copy(InputStream in, OutputStream out) throws IOException {
    int numRead = 0;
    byte buf[] = new byte[1024];
    synchronized(in) {
      synchronized(out) {
        BufferedInputStream bin = new BufferedInputStream(in);
        BufferedOutputStream bout = new BufferedOutputStream(out);
        while((numRead = bin.read(buf, 0, 1024)) != -1)
          bout.write(buf, 0, numRead);
        bout.flush();
      }
    }
  }

  protected void initMimeTypes() {
    mimeTypes = new Hashtable();
    mimeTypes.put("hqx", "application/mac-binhex40");
```

```
    mimeTypes.put("cpt", "application/mac-compactpro");
    mimeTypes.put("doc", "application/msword");
    mimeTypes.put("bin", "application/octet-stream");
    mimeTypes.put("dms", "application/octet-stream");
    mimeTypes.put("lha", "application/octet-stream");
    mimeTypes.put("lzh", "application/octet-stream");
    mimeTypes.put("exe", "application/octet-stream");
    mimeTypes.put("class", "application/octet-stream");
    mimeTypes.put("jsp", "application/jsp");
    mimeTypes.put("oda", "application/oda");
    mimeTypes.put("pdf", "application/pdf");
    mimeTypes.put("ai", "application/postscript");
    mimeTypes.put("eps", "application/postscript");
    mimeTypes.put("ps", "application/postscript");
    mimeTypes.put("ppt", "application/vnd.ms-powerpoint")
    /..
  }

  protected String getMimeType(String str) {
    String type = (String)mimeTypes.get(str);
    if(type == null)
      type = "application/octet-stream";
    return type;
  }
}
```